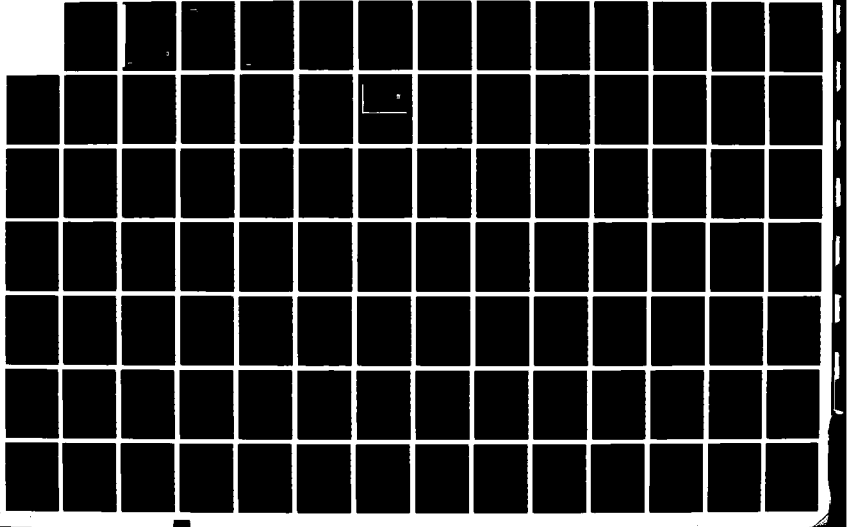
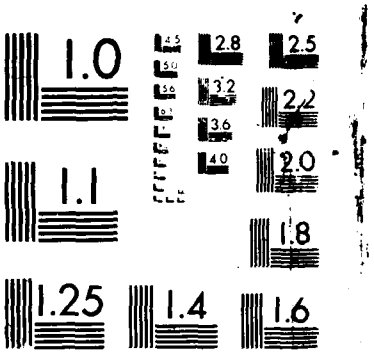
AD-A195 154 RESEARCH ON PROBLEM-SOLVING SYSTEMS(U) SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER D E WILKINS FEB 88 AFOSR-TR-88-0563 1/2

UNCLASSIFIED F49620-85-K-0001 F/G 12/9 NL





(2)

RESEARCH ON PROBLEM-SOLVING SYSTEMS

AFOSR-TR- 88-0563

Final Report

Covering the Period October 1, 1984 to February 14, 1988

February 1988

By: David E. Wilkins, Senior Computer Scientist
Representation and Reasoning Program

Artificial Intelligence Center
Computer and Information Sciences Division

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling Air Force Base
Washington, D.C. 20332

Attention: Dr. Abe Waksman

AFOSR Contract No. F49620-85-D-0001
SRI Project 7898

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025-3493
(415) 326-6200
TWX: 910-373-2046
Telex: 334486

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 88-0563
6a. NAME OF PERFORMING ORGANIZATION SRI INTERNATIONAL 6b. OFFICE SYMBOL (If applicable) AIC	7a. NAME OF MONITORING ORGANIZATION The Air Force Office of Scientific Research
6c. ADDRESS (City, State, and ZIP Code) 333 RAVENSWOOD AVENUE MENLO PARK, CALIFORNIA 94025	7b. ADDRESS (City, State, and ZIP Code) Bldg 410 Bolling AFB, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR 8b. OFFICE SYMBOL (If applicable) NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-85-K-0001
8c. ADDRESS (City, State, and ZIP Code) Bld 410 BOLLING AFB, WASHINGTON D.C. 20332	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61102F; PROJECT NO. 2304; TASK NO. A7; WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

RESEARCH ON PROBLEM-SOLVING SYSTEMS

12. PERSONAL AUTHOR(S)
DR. DAVID E. WILKINS

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 10/1/84 TO 2/14/88	FEBRUARY 1988	156

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final report for a research project which focused on artificial intelligence planning systems. The research investigated methods for representing, generating, and executing hierarchical plans that contain parallel actions. Reasoning about actions is critical to many important areas including automatic planning systems, expert consultation systems, and real-time control of robotic systems. This report describes progress in planning, including efficient technqiues for generating hierarchical and parallel plans in certain domains. This work was performed using SIPE (Sytem for Interactive Planning and Execution Monitoring) which was developed in part under this contract.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL James M. Crowley, Maj, USAF	22b. TELEPHONE (Include Area Code) (202) 767-5025
	22c. OFFICE SYMBOL AFOSR/NM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SRI International

RESEARCH ON PROBLEM-SOLVING SYSTEMS

Final Report

Covering the Period October 1, 1984 to February 14, 1988

February 1988

By: David E. Wilkins, Senior Computer Scientist
Representation and Reasoning Program

Artificial Intelligence Center
Computer and Information Sciences Division

Prepared for:

Air Force Office of Scientific Research
Building 410
Bolling Air Force Base
Washington, D.C. 20332

Attention: Dr. Abe Waksman

AFOSR Contract No. F49620-85-D-0001
SRI Project 7898

Approved:

Stanley J. Rosenschein, Director
Artificial Intelligence Center

Donald L. Nielson, Vice President
Computer and Information Sciences Division

SRI International®

Acknowledgments

Thanks to the Air Force Office of Scientific Research for supporting this work under contracts F49620-79-C-0188 and F49620-85-K-0001. The SRI International Artificial Intelligence Center provided the open, cooperative environment that encourages new ideas, a state-of-the-art computing environment, and support for the implementation of the graphical interface. Special thanks go to Stan Rosenschein, Nils Nilsson, and Michael Georgeff for technical leadership, Marietta Elliott for administrative leadership, and Paul Martin and Mabry Tyson for the computing environment. Ann Robinson helped formulate the original conception of SIPE, and Michael Georgeff has greatly influenced both the development of SIPE and the ideas expressed in this paper.

Contents

Chapter 1

Introduction

Research on planning systems sponsored by AFOSR in SRI International's Artificial Intelligence Center addresses the problem of reasoning about actions in an efficient manner. This research was begun in September 1979 under AFOSR sponsorship (SRI Project 8871, Contract No. F49620-79-C-0188). A continuation project covering the same line of work was initiated in October 1984 (SRI project 7898, Contract No. F49620-85-K-0001). SRI's main task under this program has been to develop powerful methods of representing, generating, and executing hierarchical plans that contain parallel actions. Execution involves monitoring the state of the world and, possibly, replanning if things do not proceed as expected. Since 1979, SRI has designed and implemented a system called SIPE (System for Interactive Planning and Execution Monitoring), the purpose of which is to expand our approach to this problem and demonstrate its heuristic adequacy. This report describes the entire SIPE system and the design decisions behind it, effectively summarizing all research results since 1979.

Reasoning about actions is of critical importance, and is a core problem in artificial intelligence (AI). Research results are potentially applicable to many important areas. Examples of particular interest to the Air Force include automatic planning systems (as might be used in command and control applications or to plan air operations), computational systems for aiding a flight crew, expert consultation systems, and real-time control of robotic systems (such as remotely piloted or autonomous vehicles). Other areas which depend on reasoning about actions include automatic programming (program synthesis) and natural-language understanding (which often requires knowing the plans of the speaker).

One core problem is common to all reasoning about actions. This central problem is determining how a complex world is affected by an event that occurs in the world, so that a system can reason about the world both as it was before the event and as it will be after the event. We will refer to this rather broad problem as the *frame problem*, although others define that term in many different ways, often referring to a much more specific problem (which is generally one of the many problems that must be addressed by our broad problem). The frame problem is what makes reasoning about actions inherently difficult, and what distinguishes reasoning about actions from similar problems that do not require that this problem be addressed. For example, many scheduling problems require constraints to be satisfied so that schedules can be correctly met, but do not require that the system reason about how the world changes as scheduled events occur. Scheduling problems can be very difficult and are certainly important, but they are simpler than problems that also require reasoning about the effects of actions. Recently, the term "planning" has been used by many people to refer to a broad range of problems and techniques. These range from application of control theory in robot control to the solution of constraint satisfaction problems to expert systems. We consider reasoning about how actions affect the world to be the heart of the planning problem, and restrict our use of the term "planning" to approaches that address this problem.

Recent developments — particularly during the course of work at SRI supported by this project — include efficient techniques for generating hierarchical as well as parallel plans in certain domains. Domain-independent planners yield planning techniques that are applicable in many domains and provide a general planning capability. Such a general planning capability is likely to require techniques different from those used by an expert planning in his particular domain of expertise, but it is nonetheless essential for people in their daily lives and for intelligent programs. Much related planning research involves development of logical formalisms that are theoretically correct and complete, but that have little hope of efficient implementation. By contrast, heuristic adequacy is one of the goals of our research.

SIPE has extended the classical AI planning paradigm farther than any other system. The classical definition of the planning problem assumes a state-based representation and takes as input a description of the initial state, a description of a set of actions (operators), a goal descriptor, and a set of sentences describing the domain (e.g., frame axioms, domain constraints), and requires the planner to produce as output a sequence of actions that, when

initial state, will result in the goal being achieved. The classical AI planning paradigm also supports hierarchical, nonlinear plans, and views planning as a search through the space of operator applications and plan orderings. NOAH [27] and STRIPS [6] mark the beginning of this approach, and their ideas inspired most planning research for more than a decade. Many systems developed this paradigm further — NONLIN [31], DEVISER [33], and SIPE [36] perhaps being the most important. NOAH is the ancestor of this approach since it produces nonlinear, hierarchical plans, but the root of the solution to the frame problem in all these systems is found in STRIPS. The STRIPS assumption [34] is at the heart of the efficiency achieved by the above systems, as well as the cause of many of their limitations. Some of these systems, notably SIPE, modify the STRIPS assumption radically enough to avoid some of its pitfalls.

SIPE has become a state-of-the-art planning system during its development at SRI under AFOSR sponsorship. It provides a domain-independent formalism for describing a domain at different levels of abstraction, including both actions that can be taken and goals that can be achieved. It extends the classical AI planning approach by reasoning about resources, posting constraints, and using a deductive causal theory to represent and reason efficiently about different world states. SIPE retains much of the efficiency of the STRIPS assumption while avoiding some of its disadvantages through the use of the above mechanisms. The system automatically, or under interactive control, generates [possibly nonlinear] plans containing conditionals to achieve goals in an initial situation. It can intermingle planning and execution, and accepts arbitrary descriptions of unexpected occurrences during execution and modifies its plan to take these into account.

The difficulty of the planning problem should not be underestimated. Chapman [4] has shown that determining the truth of a proposition in a nonlinear plan (i.e., a plan in which some actions are unordered with respect to each other) is NP-hard, even with a very restricted representation, as long as the representation allows actions whose effects depend on the input situation (as SIPE does). Given the extensions SRI has made to previous planners, there are several combinatorial problems, in addition to the truth criterion, that must be solved. SIPE's unification problem also becomes combinatorial with its reasonably powerful constraints on variables. The problem of parallel interactions, the system's resource allocation problem, the deduction of context-dependent effects, the search through the space of possible plans, and the search through modified plans during replanning are all combinatorial. Restrictions on

SIPE's representations have been combined with heuristics and algorithms in order to provide a useful planner that addresses all the above problems while remaining efficient.

While we will discuss the theoretical foundations of SIPE in this report, they will not be our major focus. The major focus will be the communication of the planning technology developed during work on SIPE to other planning researchers. We explain the basic assumptions underlying SIPE (and the reasons for them), and the kinds of domains for which this planning style is and is not suited. The major modules in SIPE will be described so that others can understand how they work and why they were designed as they now exist. These modules include the plan critics, the execution monitor and replanner, the algorithm for determining the truth of a formula, the capability for specifying a deductive causal theory of the domain, resource allocation and constraint satisfaction, and the newly implemented capabilities for reasoning about numerical quantities. The heuristics developed and the tradeoffs considered in making them are of primary interest and constitute the major part of this report.

Chapter 2

Basic Assumptions and Limitations

SIPE is a Zetalisp program that has been developed over several years and runs on the Symbolics 3600 family of machines. It has produced correct plans in several different domains, including the standard block world, several extensions of the block world (e.g., one with blocks of different weights in which the robot consumes fuel as a function of the weight it is moving), cooking, aircraft operations, travel planning, construction of objects in a machine shop, and an indoor mobile-robot domain. Block-world problems that permit more than one block to be on top of another are solved in one or two seconds on a Symbolics 3600, providing a scale for our claims of heuristic adequacy.

The robot domain will be used in several examples in this report, so a brief description of it is in order. SRI International has built a mobile robot, Flakey, which is used as a testbed for several projects and roams the halls of the Artificial Intelligence Center (AIC). In an effort to provide Flakey with a high-level planning capability, we have encoded in SIPE a domain consisting of five rooms connected by a hallway in the AIC, the robot itself, and various objects. The rooms were divided into 35 symbolic locations, and the initial world is described by 222 predicate instances. The description of possible actions in SIPE includes 25 action-describing operators and 25 deductive rules. The operators use four levels of abstraction in the planning process, as described in Chapter 7. The planner produces primitive plans that provide commands, executable by Flakey, for controlling the robot's motors.

Planning to such a low level of abstraction consumes considerable computational re-

sources. To solve a problem requiring the robot to retrieve an object from one room and deliver it to another typically requires the planner to generate hundreds of goal nodes (just to generate one plan, not to search through alternatives), yet SIPE takes about 30 seconds to completely formulate such a plan. By taking advantage of the system's ability to intermingle planning and execution, a plan can be ready for execution in only 9 seconds. This is acceptable performance as the robot requires several seconds to move down the hall. Previous classical planners have not been tried on problems of this size, in most cases because such problems cannot be effectively handled. Many planners that use frame axioms or circumscription instead of the STRIPS assumption have combinatorial problems and currently have no hope of producing a plan of this complexity in a matter of seconds. We know of no planning system that approaches the speed of SIPE on a problem as complex as this.

SIPE builds upon the classical AI planning work exemplified by Sacerdoti's NOAH, Tate's NONLIN, and Fikes and Nilsson's STRIPS. While its representations and algorithms have almost nothing to do with these systems, SIPE accepts the classical definition of planning given in Chapter 1 (implying a lack of reactivity during plan time), and supports hierarchical, nonlinear plans. The system takes as input a description of the initial state in a restricted form of first-order predicate calculus together with a sort hierarchy that encodes static knowledge, and a description of actions, goals, and domain knowledge in language provided by SIPE. The system automatically, or under interactive control, generates plans (i.e., sequences of known actions) to achieve the given goals and supports replanning after unexpected occurrences during execution. SIPE makes the closed-world assumption: any negated predicate is true unless the unnegated form of the predicate is explicitly given. While this is not critical, it makes the specification of domains much easier since there may be an enormous number of predicates that are not true and it may not be possible to summarize all these concisely within our representation.

One of the primary design principles of the system has been heuristic adequacy. To obtain a useful planner, we have opted for restrictions on representations and algorithms and heuristics that will make the system heuristically adequate without destroying its usefulness. The goal is to have a representation that is rich enough so that many interesting domains can be represented (an advantage of logical formalisms), but this goal must be measured against the system's ability to deal with its representations efficiently during the planning process. Tradeoffs considered in creating these restrictions will be described throughout this

report as they impact almost every part of the planner. SIPE can be viewed as providing tools that are useful for solving planning problems. These tools may not themselves be sound and complete, and they may not match exactly with the user's problem domain, but with a proper encoding of the domain these tools can be powerful aids in producing correct plans efficiently.

2.1 Important Features

Two important and common, but complex, features of planning systems are central in SIPE: planning at different levels of abstraction (also known as hierarchical planning) and nonlinear plans (including possibly parallel actions). Planning in abstract spaces is necessary in real-world domains, since it helps avoid the tyranny of detail that would result from planning at the most primitive level. The planner can significantly reduce the search space by forming abstract plans and then expanding these into more detailed plans. Chapter 7 contain. a general discussion of hierarchical planning that enumerates its many uses in various systems. It describes a problem, uncovered during an application of SIPE, that applies to all hierarchical planners, and presents several solutions implemented in our system.

Nonlinear plans are also necessary for most real-world domains. Even for single agent domains, correct plans will be achieved only by ordering actions during the planning process. Actions can often remain unordered until such time as the planner discovers the order it wishes to impose. In addition, SIPE allows parallel actions, meaning that actions can remain unordered in the final plan and the system will assure that no harmful interactions occur. A linear planner could search the space of all possible orderings without explicitly representing unordered actions, but this requires exponentially more backtracking. Furthermore, real-world domains are often multieffector or multiagent (e.g., having two robot arms to construct an object, or two editors to work on your report), and the best plans should use these agents in parallel whenever possible. For these reasons, SIPE supports nonlinear plans; however, this incurs a substantial cost as it leads to an NP-complete truth criterion under the conditions described by Chapman, although this is almost certainly preferable to the exponential search of a linear planner. The way in which SIPE's truth criterion avoids this complexity is described below and in Chapters 4 and 5.

A planning system that allows parallel actions must be able to reason about how actions

interact with one another, since interference between parallel actions may prevent the plan from accomplishing its goal. This is a major problem for planning systems, and distinguishes planning from much of the work in program synthesis, since the goal there is often a strictly sequential program. SIPE solves the interaction problem by extending the idea of *plan critics* introduced by NOAH. After each level of planning, the system may produce invalid plans, but it then applies critics which check for problems such as parallel interactions and unsatisfiable constraints. If problems are detected by the critics, *solvers* are applied to modify the plan, possibly adding ordering constraints to the parallel actions. Solvers in SIPE are more powerful than those in previous classical planners, as they use the replanning actions of the execution monitor to modify plans, possibly removing subplans in order to make more optimal plans. The critics and solvers are discussed further in Chapter 9. Because invalid plans are produced and then corrected, the idea of defining conditions that make operators sound, as Lifschitz attempts to do for STRIPS [17], is not directly applicable.

One of SIPE's primary contributions is the use of resource reasoning to help solve the parallel interaction problem (described in Chapter 10). Much of the work done by plan critics in previous planners, e.g., the resolve-conflicts critic in NOAH, is accomplished by resource reasoning in our system. Both reusable and consumable resources are supported. When actions declare objects as resources, the system can quickly detect resource conflicts and linearize the contending parallel actions. While the concept of resources in SIPE is limited, it is nevertheless quite useful both in the representation of domains and in finding solutions efficiently.

The nucleus of SIPE's quest for heuristic adequacy is its efficient truth criterion. The latter is based on the STRIPS assumption, which is also used by many seminal planners such as STRIPS, NOAH, and NONLIN. The (strict) STRIPS assumption is that no predicate will change its truth value when an event takes place unless the event explicitly lists that predicate on its add or delete lists. As we shall see, this strict assumption adversely affects the specification of operators (i.e., a planner's representation of actions or events), making them awkward or impossible to describe, especially as domains grow more complex. SIPE alleviates these problems through its use of constraints, resources, and domain rules. Domain rules, which are described in detail in Chapter 6, are used to deduce the effects of an event that are conditional on the current situation and cannot therefore be mentioned in add or delete lists. They permit effective representation of a *causal theory* of the domain, similar

to that advocated by Dean [5]. By allowing knowledge of cause-and-effect relations to be specified independently of the operators, both the operators and the planning process are simplified. Since conditional effects are deduced, operators are applicable over a much wider range of situations. This makes it much easier for the user to express his domain knowledge as SIPE operators.

The truth criterion also provides a mechanism for circumventing the poor performance that is caused by the need to solve a NP-complete problem when determining the truth of predicates over parallel actions. The system allows the user to distinguish between main effects and side effects of an action, and does not consider all possible shuffles of parallel actions when matching predicates that occur in side effects. The system guarantees correctness over parallel actions only when matching predicates that are given as main effects. For predicates occurring in the side effects of a parallel action, the truth criterion proves that there is one possible ordering of the parallel actions that makes the predicate true without enforcing that order. There are mechanisms for preventing the system from making contradictory assumptions about different orderings as planning proceeds. This has proved to be a useful compromise that provides the user with enough tools to produce useful plans efficiently. The truth criterion and the tradeoffs it involves are described in detail in Chapter 5.

One of SIPE's most important advances over previous domain-independent planning systems is its ability to post constraints in order to construct partial descriptions of unspecified objects. This ability is important both for domain representation and for finding solutions efficiently (since decisions can be delayed until partial descriptions provide more information). Chapter 4 describes the constraint language, explains how it is incorporated into the system, and compares it with other systems. Almost no previous domain-independent planning systems have used this approach (e.g., NOAH cannot partially describe objects), and domain specific systems which use constraints, such as MOLGEN [29], generally deal with constraints that are also domain specific.

The use of constraints can be viewed as extending the idea of "least commitment" planning. The idea behind least commitment planning is to delay decisions until you have as much useful information as possible for making them. This term was first introduced by Sacerdoti in connection with NOAH. NOAH avoided commitments by using nonlinear plans to delay ordering decisions. NOAH also did not backtrack, and this feature of the system has been associated with the term "least commitment" by some people, but these are actually

two completely separate issues — SIPE does backtrack but still follows a least commitment philosophy. While NOAH delayed ordering decisions, a more powerful representation can delay decisions in many other situations. SIPE does exactly this by allowing constraints on its planning variables, whose eventual instantiations can then be chosen more intelligently than in a system where a variable is either unbound or instantiated.

SIPE has mechanisms for reasoning about numerical quantities, both continuous and discrete, which provides the basis for reasoning about producible and consumable resources, as well as limited forms of temporal reasoning (e.g., specifying constraints on the starting time of an action). The same representations and algorithms work for both these tasks because time is considered to be a type of consumable resource — namely, one that can be consumed but not produced, and whose consumption in the course of parallel tasks is nonadditive. Numerical reasoning is integrated within the existing framework of adding constraints to planning variables, allowing the system to employ all its standard algorithms to solve numerical problems.

Our research has not addressed the issue of intelligent control of the search process. In part, this is because different searching algorithms will function best in different domains. Certainly an intelligent search procedure will need domain-dependent heuristics. SIPE implements a straightforward depth-first search with chronological backtracking for generating plans automatically that permits interleaving of planning and execution. Unlike its predecessors, SIPE provides two other capabilities: a context mechanism that allows easy access to alternative plans, and interactive control of the search. The former allows the user to implement any domain-dependent search strategy he chooses, including a best-first search. The latter allows the user to watch and, when he wishes, guide and/or control the planning or replanning process.

In real-world domains, things do not always proceed as planned, making it necessary to monitor the execution of a plan and to replan when things do not go as expected. In complex domains it becomes increasingly important to use as much as possible of the old plan, rather than begin again. SIPE's execution monitor accepts arbitrary descriptions of unexpected events, and is able to determine how they affects the plan being executed. In many cases, it is able to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. It is also capable of shortening the original plan when serendipitous events occur.

The most important features of the system are itemized below:

- Domain independence
- Different abstraction levels
- Nonlinear actions
- Powerful plan critics
- Resource reasoning
- Efficient truth criterion
- Deduction of context-dependent effects of actions
- Posting of constraints
- Numerical reasoning
- Interactive or automatic search
- Replanning

2.2 SRI's Graphical Interface

During the past year, SRI International has greatly enhanced a user's ability to use SIPE by developing (with internal SRI funds) a completely separate module called the SIPE Graphics Interface. All the planning mechanisms in the SIPE Planning System were implemented under AFOSR sponsorship. As so implemented, SIPE is run by calling functions from the terminal and by producing what is essentially teletype output on a terminal. The planning system and graphical interface together make an integrated system, shown in Figure 2-1 displaying a plan from the robot domain, that is much easier to use.

The SIPE Graphics Interface allows a user to control the planning system from menus, to display data structures from menus, and to view plans as graphs on the screen. The graphical display of plans allows the user to change the sizes of nodes in the graphical display, change types of nodes displayed, and change slots displayed with each node. The graphical output window is scrollable, so the window can be moved left, right, up, or down over the displayed plan. The displayed nodes are mouse-sensitive, and clicking them will cause them to be displayed in their entirety. The SIPE Graphics Interface makes use of software developed

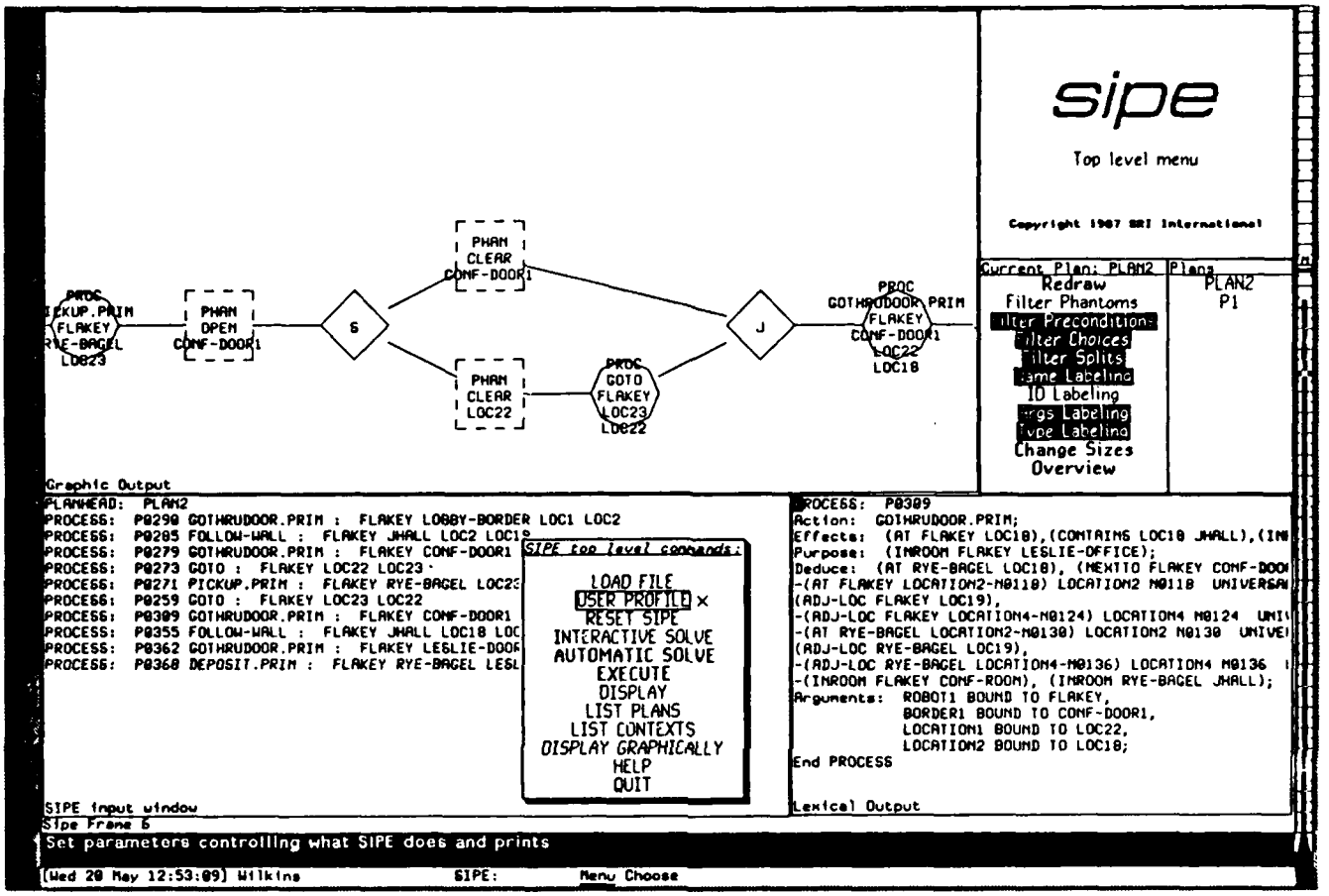


Figure 2.1: SIPE's Graphical Interface

at SRI for displaying graphs on scrollable windows as implemented in the SRI X-Y Window Package. Most of this module, which is not unique to SIPE, was written primarily by Josh Singer, Mabry Tyson, and Paul Martin.

2.3 Limitations

Classical planners are useful for the kind of planning we do in our daily lives when we need to stop and think about what to do next. This can often be thought of as a type of means-end analysis. For example, when running several errands, one usually stops to plan the order in which they will be done. Classical planners are not well-suited for large scheduling tasks (for which people may use linear programming techniques), large constraint satisfaction problems, highly dynamic worlds, or sophisticated reasoning about other agents. Besides lacking powerful computational techniques for solving the former two problems, most classical planners, and SIPE in particular, incorporate heuristics for dealing with nonlinear plans that are more suited for problem-solving tasks (where the object is to find any acceptable solution) than for scheduling or optimization tasks (see Chapters 5 and 9). While limitations are often ascribed to classical planners incorrectly, they do have many shortcomings. In this section we summarize many of the restrictions incorporated into the SIPE system.

Like other classical planners, SIPE assumes a state-transition approach to representing a dynamic world. (Although the world is assumed to be not so dynamic that it will change significantly during plan time.) Actions change the world from one discrete state to another. For the most part, operators, actions, time, and states are all discrete. SIPE does extend previous classical planners by allowing conditional plans, replanning after unexpected events, and providing a capability for reasoning about continuously changing numerical quantities as part of reasoning about consumable resources (see Chapter 10). The latter permits some rudimentary forms of reasoning about time and continuous quantities, but sophisticated reasoning about time and modeling of dynamic processes are not possible within our present framework. (Of course, very few artificial intelligence programs have addressed these latter problems.) Unless consumable resources are employed, time is not represented explicitly, since the ordering links in the procedural network provide the necessary temporal information.

Having discrete operators and actions means that the effects of an action occur instantaneously as far as the system is concerned. This applies to a given abstraction level; using

hierarchical planning, the system can order the effects that occur at a lower level of detail. While the deductive causal theory could deduce effects that depend on time, the system is not suitable for highly dynamic worlds. It is also not designed to monitor the world as it is planning, and therefore cannot react immediately to a changing environment.

Proposals for reasoning about the beliefs and knowledge of other agents generally specify more sophisticated logics, particularly modal logics [10]. These capabilities cannot be represented in SIPE, because it uses a restricted form of standard first-order logic to represent dynamic relationships. Many other logics (e.g., temporal and dynamic logics) have also been proposed for planning, and they often provide more expressive power than is found in classical planners. However, they suffer from inherent computational difficulties, the need to write many axioms with all the details right, and possibly other problems such as unintended models [11] (in nonmonotonic systems) or the need to compute all possible effects an action might have. With an expressive logic there is generally a need to specify axioms to deduce that all things not mentioned have stayed the same (unless the STRIPS assumption or something similar is employed).

Another major limitation of classical planners is that they assume complete and correct knowledge of the world. This is, of course, *unrealistic* in the real world, although there are certainly useful problems to be solved in domains where the state of the world is known. SIPE alleviates this problem somewhat by allowing predicates to be specified as unknown, but the system does no sophisticated reasoning about uncertainty. This limitation is also extended into the execution monitoring and replanning modules, which assume correct information about unexpected events. The limitation avoids many difficult problems, the most important of which is generating the high-level predicates used by SIPE from information provided by the sensors. This appears to be the most critical issue in enabling a high-level planner such as SIPE to control a mobile robot.

Although many of the limitations described above are quite severe, there are nevertheless useful problems that can be addressed within these limitations. Examples include the planning of tasks for an indoor mobile robot, planning a travel itinerary, and producing a process routing for a manufacturing facility. In return for accepting these limitations, we get an efficient system than can solve these problems (which would otherwise involve combinatorial explosions) in reasonable amounts of time.

Chapter 3

SIPE and its Representations

As we have seen, SIPE accepts the classical definition of the planning problem and takes as input a description of the initial state, a description of a set of actions, a goal descriptor, and a set of sentences describing the domain. We use the term *operator* to refer to the system's representation of actions or abstractions of actions that may be performed in the domain. To meet its goals of efficiency, SIPE uses both a frame-based representation and first-order logic to describe the domain. In addition, an operator description language is provided for describing operators in such a way that SIPE's truth criterion (see Chapter 5) can be used on all plans produced from these operators. The operator description language was designed to be easy to understand (to enable graceful interaction) while being more powerful than those found in previous domain-independent planners. Furthermore, rules in the deductive causal theory (see Chapter 6) can also be expressed in the language.

Figure 3-1 shows how all these different aspects of the system fit together. It depicts a conceptual division of the planning system into different modules, primarily for expository purposes — there is not always a sharp demarcation in the actual code that separates these modules. The remainder of this report will describe each of these modules.

The flow of control depicted in the figure indicates how the modules interact with each other in the planning system as a whole. The search algorithm controls the planner, and constructs plans by using the interpreter to apply operators to expand existing plans. In this chapter, we describe the operator description language, plans, and the basic representations they depend on. To expedite graceful interaction, plans in SIPE are represented as procedural

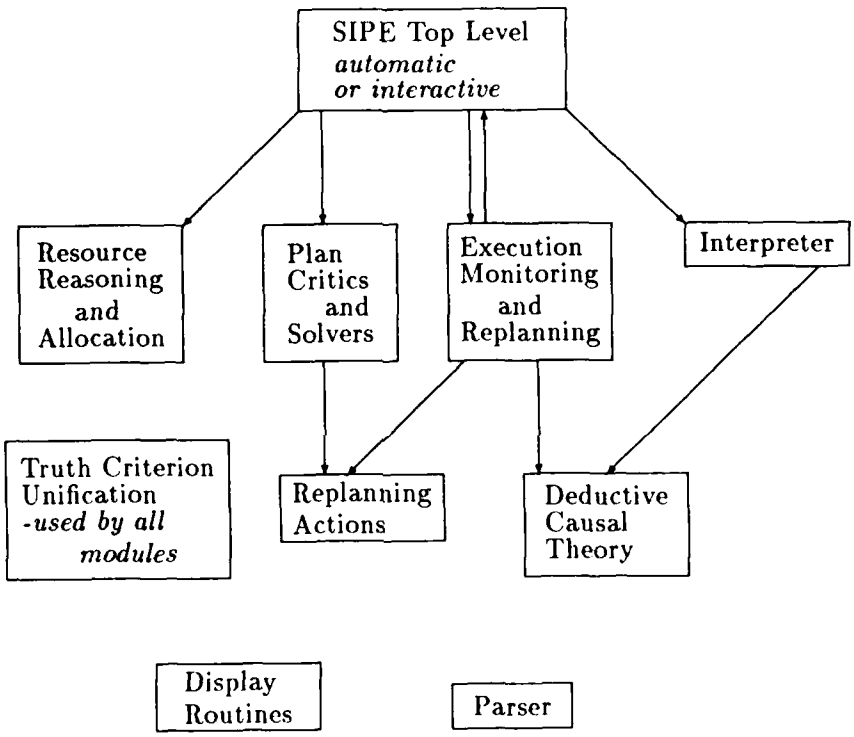


Figure 3.1: SIPE Modules and Flow of Control

networks [27], with temporal information encoded in the predecessor and successor links between nodes. The description of operators avoids many problems suffered by previous domain-independent planning systems; because we use a deductive causal theory, most effects of actions can be deduced in a context-dependent manner so that they do not need to be specified in the operators.

The truth criterion determines whether a formula is true at a particular point in time. All modules of the system depend upon it, and the system's computational efficiency depends directly on the efficiency of the truth criterion. Chapter 5, the most technical in this report, describes it in detail, explaining the heuristics that avoid solution of an NP-complete problem. Many design decisions in other modules were chosen to take advantage of the efficient truth criterion. The truth criterion relies on the unification procedure; SIPE is a constraint-posting planner, so unification involves reasoning about constraints and deciding how much of the global constraint network to analyze. Chapter 4 describes SIPE's constraints and the unification procedure.

The search algorithm must balance the use of resource reasoning and use of plan critics with the application of operators. If the global constraint network is checked too frequently, the planning will be unnecessarily slow. If it is not checked frequently enough, the system may spend its computational resources developing plans that will later be found to contain inconsistent constraints. SIPE's search algorithm (described in Chapter 8) makes certain tradeoffs on these issues, but allows interactive control by the user for added flexibility. The search algorithm must also be concerned with the relationship between different levels of abstraction in the plan. Much of the system's power comes from using these levels, and the issues involved are discussed in their own chapter on hierarchical planning.

The plan critics are responsible for finding problems in the plans produced and correcting them. Primarily, this involves checking whether the global constraint network is satisfiable, finding resources conflicts, checking with goals are already true, and finding problematic interactions between parallel actions. Most of the critics are described in Chapter 9, but the ones that do resource reasoning are especially important because of their innovations and are discussed separately in Chapter 10. The critics make use of the replanning actions that are part of the execution monitor. By so doing they modify plans, sometimes removing subplans in order to make a more optimal plan. This represents a significant advance over previous classical planners.

When planning is finished, the search algorithm relinquishes control to the execution monitor. This module (described in Chapter 11) accepts descriptions of arbitrary unexpected occurrences. It then determines how these occurrences affect the plan being executed, possibly modifying the plan by removing certain subplans and inserting certain goals. When the resulting plan contains unsolved goals, the execution monitor again calls the search algorithm to expand this plan. Both the plan critics and the execution monitor use the replanning actions to alter existing plans. The execution monitor makes use of the *plan rationale* that SIPE has encoded in the plan. (The rationale for an action in a plan is "why" the action is in the plan.) This is needed for determining how long a condition must be maintained, what changes in the world cause problems in the plan, and what the relationship is among actions at different levels of abstraction. The replanner can then use the rationale to decide what subplans to remove or modify when unexpected events have occurred. The current chapter explains how operators provide the information needed by the plan rationale.

The parser and display routines will not be described elsewhere in this report. The parser reads in the perspicuous formalism accepted by SIPE and produces the internal structures the planner uses. All operators and deductive rules presented in this report are given in the syntax accepted by the parser. An initial description of a domain includes specification of the sort hierarchy describing the objects in the world, predicates stating what is true in the world, operators stating which actions may be taken, operators describing deductive rules, and procedural networks for the problems to be solved. The last three are all described using the operator description language described below.

3.1 Representation of Domain Objects and Relationships

Domain objects and their invariant properties are represented by nodes linked in a hierarchy. This permits SIPE to incorporate the advantages of frame-based systems (primarily efficiency), while retaining the power of the predicate calculus for representing properties that do vary. Invariant properties do not change as actions planned by the system are performed (e.g., the size of a truck does not change when it is driven). Each node can have attributes associated with it and can inherit properties from other nodes in the hierarchy. The values of attributes may be numbers, pointers to other nodes, key words that the system recognizes, or any arbitrary string (which can be used only by checking whether it is equal to another

such string). Planning variables contain constraints on the values of attributes of possible instantiations.

A restricted form of first-order predicate calculus is used to represent properties of domain objects and the relationships among them that may change with the performance of actions. This calculus is also used to represent invariant relationships in the domain, although relationships that can be represented as unary or binary predicates would normally be placed in the sort hierarchy. Predicate names can be specified as invariant, in which case the system avoids the inefficiency of applying the truth criterion to them. This same calculus is, of course, used to describe goals, preconditions, and effects in the operator description language. Quantifiers are allowed whenever they can be handled efficiently — certain universal quantifiers are permitted in effects (but not preconditions), and certain existential quantifiers can occur in the preconditions (but not effects). Disjunction is also not allowed in effects. These restrictions result from the way the truth criterion solves the frame problem.

3.2 Operator Description Language

The operator description language was designed to provide an easily understandable way to specify operators. Operators represent the actions, at different levels of abstraction, that the system may perform in the given domain. The primary representational task of an operator is to describe how the world changes after the action it represents is executed. A brief summary of our solution to the frame problem (presented in detail in Chapter 5) should be enough to understand the representation of operators. SIPE makes the assumption that the world stays the same except for the effects explicitly listed with each action. However, the system deduces many of these effects from the deductive causal theory of the domain. So operators must explicitly list only effects that are required to trigger all the necessary deduced effects, thus relieving the operators of much of their representational burden. The effects explicitly listed in the operator must, of course, occur in every situation in which the operator might be applied, while the deduced effects may be conditional on the situation.

In addition to effects, operators contain information about the objects that participate in the actions, the constraints that must be placed on them, what the actions are attempting to achieve, how actions in this operator relate to more or less abstract descriptions of the same action, and the conditions necessary before the actions can be performed (their preconditions).

Operator: Fetch
Arguments: robot1,object1,room1;
Purpose: (Holding robot1 object1);
Precondition: (Inroom object1 room1);
Plot:
    Goal: (Inroom robot1 room1);
        Protect-until: (Holding robot1 object1);
    Goal: (Nextto robot1 object1);
    Process
        Action: Pickup;
        Arguments: robot1, object1;
        Effects: (Holding robot1 object1);
End Plot End Operator

Figure 3.2: SIPE Robot World Operator

In addition, the planner must encode the rationale behind the plan so that the replanner can make informed decisions about plan modification. Much of the knowledge about plan rationale is provided in the operators. Many features combine to make SIPE's operator description language an improvement over operator descriptions in previous systems. These features will be presented by discussing the sample operator given in Figure 3-2. The Fetch operator comes from a fairly high abstraction level in the mobile robot domain and describes the fetching of an object from another room.

The *purpose* of an operator determines which goals the operator can solve (as well as in the plan rationale), and its *precondition* dictates in which situations the operator can be applied. Applying an operator involves interpreting its *plot* as a subplan for achieving a goal. The *arguments* of an operator are templates for creating planning variables and adding constraints to them. The operator's preconditions and purpose are both encoded as first-order predicates on the arguments of the operator, which can be variables or objects in the domain. Each of these will be discussed below using the Fetch operator as an example. Other properties of operators are used by deductive rules, but these will be described later.

3.2.1 Arguments

In the Fetch operator, robot1, object1, and room1 are variables that are constrained (by virtue of their names) to be in the classes robots, objects, and rooms respectfully. Besides this automatic posting of class constraints, the listing of arguments in an operator can convey considerable additional information about resources and constraints. Arguments can be specified as resources, in which case the system treats them as reusable resources (as described in the chapter on resources). Constraints will also be described in their own chapter, but the following syntax, which SIPE accepts, gives an indication of the variety of phrases that can occur after a variable in the arguments slot:

```
CLASS EXISTENTIAL
CLASS UNIVERSAL
CLASS <any defined class>
OF TYPE <any defined class>
IS NOT <any defined instance>
IS NOT OF TYPE <any defined class>
WITH <any attribute name> <any attribute value>
WITH <any attribute name> GREATER THAN <any numerical value>
WITH <any attribute name> LESS THAN <any numerical value>
SAME AS <any previous variable>
IS <numerical function specification>
IS <numerical range>
IS <number>
IS CURRENT VALUE OF <continuous numerical variable>
IS PREVIOUS VALUE OF <continuous numerical variable>
```

The ability to post and use constraints like the ones above is a powerful tool that was not present in previous domain-independent planners.

3.2.2 Preconditions

An operator's precondition must be true in the world state before the operator can be applied. In the Fetch operator, the precondition is used to instantiate room1 to the room which currently contains object1. The concept of precondition here differs from its counterpart in some planners, since the system will make no effort to make the precondition true -- a false precondition simply means that the operator is inappropriate. Conditions that the planner

should make true (which may be referred to as *preconditions* in other planners) are expressed as goals in the plot of the operator. Having both subgoals and preconditions that are not achieved as subgoals gives SIPE the flexibility to encode metaknowledge on *how* to achieve goals (i.e., do not try to achieve preconditions). This encoding of metaknowledge is further enhanced by the use of *choiceprocess* and *process* nodes as well as goal nodes in the plots of operators. While goal nodes implicitly state that any possible operator should be used to achieve the goal, choiceprocess nodes specify which set of operators should be used to achieve the goal, and process nodes specify one particular action that must be used (and therefore do not produce backtracking points).

Preconditions in SIPE are useful for a number of reasons. First, it is plausible that some domains may have actions that will work in certain (possibly undesirable) situations, but that one would not want to work to achieve such a situation for the sake of performing that action. SIPE can easily represent this, whereas a planner that tried to achieve all preconditions might try to make the situation worse in order to apply its "emergency" operators. Second, SIPE's preconditions are useful for connecting different levels of abstraction. The precondition of an operator might specify that certain higher-level conditions must be true, while the operator itself specifies goals at a more detailed level. This provides an interface between two different levels of abstraction that was not present in NOAH. Third, preconditions are included in the plans produced because they represent part of the plan rationale (see below) and play a crucial role in the replanner.

3.2.3 Plots

The plot of an operator provides step-by-step instructions for performing the action represented by the operator. When expanding a plan to a lower level of detail, SIPE uses the plot as a template for generating nodes to insert in the plan. (Several types of nodes not mentioned in the plot are also generated; for example, nodes denoting preconditions.) Because of this isomorphism with plans, plots are also represented as nodes in procedural networks. The plot may be at the same level of abstraction as the purpose of the operator (e.g., in the standard block world the level of description never changes), or it may use a more detailed level of abstraction.

To provide for encoding metaknowledge of how to achieve goals, the plot of an operator

can be described in terms of *goal* nodes which require a certain predicate to be achieved, *choiceprocess* nodes which require that one of a certain set of operators be applied to solve a certain predicate, and *process* nodes which require a specific operator or primitive action to be applied. These convert directly into plan nodes of the same type when an operator is applied. Unlike process nodes, the other two types may be turned into *phantom* nodes inside a plan if their goal predicate is already true in the world state without taking any action. Using a process node in the plot emphasizes the actual action being performed, while using a goal node stresses the situation to be achieved.

During planning, an operator is used to expand an already existing goal, choiceprocess, or process node in the plan to produce a more detailed plan at the next planning level. For example, if the plan contains the goal of having the robot hold an object, then the Fetch operator may be applied, and it would generate two goal nodes (one for getting into the same room as the object, and one for getting next to the object) and a process node (for picking up the object) at the next planning level. It would also generate a *choice* node and *precondition* node in the plan that are important to search control and replanning respectively. The choice node denotes that there are other choices for achieving the holding goal, and becomes part of the context that allows constraints posted on variables as a result of this operator application to only be considered when this choice is part of the current context. The precondition node helps encode the plan rationale to guide the replanner. It records the fact that the precondition of the Fetch operator was expected to be true at a particular point in the plan.

Many previous domain-independent planners required "add" and "delete" lists to be provided in operators. In SIPE, this is not necessary because the deductive causal theory deduces most of the effects of the nodes in a plan. For example, Figure 3-3 shows the Puton operator from the standard block world. Note that nothing is said in the effects of any plot node about when a block is clear or not clear, or when a block is not on another block. The causal theory for the blocks world deduces all these effects as appropriate to the situation, and handles a richer block world than does NOAH (e.g., several small blocks can be on top of a big block, requiring the moving of several blocks in order to clear the big block.) While adding significantly to the computational complexity, deduced effects make the description of operators much simpler and permit them to be applied in a much wider range of situations, since the deduced effects can be conditional. This also makes operators easier to add, modify, and debug. There is a cost involved in addition to the computational complexity — namely,

the user must provide a correct causal theory and debug it. The manner in which effects are deduced is one of the major contributions made by SIPE. Chapter 6 discusses how the causal theory is expressed, how the deductions are controlled, how the system integrates the causal theory, and the importance of deduced effects to overall system performance.

SIPE distinguishes between main effects and side effects of an action. This distinction is of primary importance in both the handling of parallel interactions and the nonlinear truth criterion, both of which will be discussed in later chapters. All effects that are deduced are considered to be side effects. Effects that are provided by the goals of the problem or are introduced directly by operator applications are considered to be main effects. Flexibility is achieved by allowing plot nodes in operators to specify side effects as well as effects (which are assumed to be main effects). If an operator specifies some predicate as a [main] effect, the deductive causal theory will not deduce it as a side effect. Thus, the writer of the operators has complete control over what which effects will be main effects and which side effects. In practice, the default of using all deduced effects as side effects and all listed effects as main effects has proven satisfactory in all domains encoded in SIPE.

3.3 Plan Rationale

The plan rationale describes "why" the plan is the way it is, so that the replanner can modify it appropriately. While SIPE provides more flexibility in specifying the rationale behind a plan than many domain-independent planners, it does not improve on NONLIN and O-PLAN [32] in this regard. The primary tasks of the plan rationale in SIPE are to encode why nodes are in the plan, how to group nodes together into subplans that accomplish a goal, how long the truth of a particular goal must be maintained, and how different abstraction levels connect. These tasks are performed by precondition nodes, protect-until links, and purposes.

Precondition nodes help solve the first two tasks. They encode the assumptions made by the planner when inserting a subplan into the plan. Furthermore, they can be used to precisely access this subplan from any point in the plan. Because precondition nodes are copied down to other planning levels, one can follow the ancestor links (described later) to the point where the precondition node was first introduced and use descendant links to determine the subplan formed by the application of the operator containing the precondition. Since preconditions are not achieved as subgoals, they are not reachieved by the replanner

Operator: Puton
Arguments: block1, object1 Is Not block1;
Purpose: (On block1 object1);
Plot:
Parallel
    Branch 1:
        Goals: (Clear object1);
    Branch 2:
        Goals: (Clear block1);
End Parallel

Process
Action: Puton.Primitive;
Arguments: block1,object1;
Resources: block1;
Effects: (On block1 object1);

End Plot End Operator

Figure 3.3: SIPE Block World Operator

when they become unexpectedly false. Such a false precondition merely means that the subplan determined by this precondition is invalid and should be removed.

When a node is planned to a greater level of detail by applying an operator, the expansion may consist of many nodes. It is important to ascertain when the effects of the higher-level node become true in the more detailed expansion. SIPE uses the purpose attribute of an operator to determine this. The higher-level effects are copied to whichever node in the expansion achieves the purpose of the operator, and the rationale for that node being in the plan is that it achieves the higher-level goal. It therefore inherits the protect-until attribute of the higher-level node, recording the fact that its effects must be maintained until the time specified in the higher-level node. A protect-until slot can have the atom PURPOSE as its value, denoting that the given node is the main purpose of the plan, not preparation for some later action. If the operator does not have a purpose attribute, or it is not listed as an effect explicitly in any node of the expansion (this will often happen when the purpose is at a higher level of abstraction than the plot), then the default is to copy the effects down to the last node of the expansion. In NOAH the assumption was that the last node of an expansion achieved the main purpose, so the effects were always copied down to that node. SIPE therefore provides additional flexibility – for example, operators that include some "clean-up" or normalization after accomplishing their goal can be represented correctly.

SIPE also keeps track of the rationale for each node that is not required for achieving some higher-level goal. Such a node is put in a plan for the purpose of preparing some later action at that level, and this intent must be recorded so the planner can maintain the effects of the node until its purpose is achieved. Nodes within the plot of an operator may specify protect-until attributes that indicate that their main effects should be maintained until the protect-until condition is achieved. If no protect-until is specified, the default is that the effects are protected until the action which achieves the higher-level purpose of the operator. Another reasonable default would be to protect the effects only until the next action. Since the system provides flexibility in this specification, the default can be tailored to each domain. In the Fetch operator, the goal of having the robot in room1 should be maintained until the robot is holding the object it is trying to fetch. If the default were to maintain an effect only until the next action, then it would be necessary to include the protect-until shown in Figure 3-2. Without it, the replanner would think that nothing was wrong if the robot left the room after getting next to the object but before picking it up. SIPE's ability to represent

protect-until attributes explicitly provides flexibility, and represents an advance over NOAH. HACKER [30] and many systems based on logic are completely flexible in this manner.

3.4 Plans

Plans are specified in SIPE by giving a pointer to the *planhead* node at the beginning of the procedural network and a context. The network contains choice points from which alternative plans branch, and the context indicates which branches should be taken in perusing the specified plan. Let us summarize the types of nodes and links that can appear in the procedural networks that represent plans in SIPE. Planhead nodes mark the beginning of each plan, and contain in their list of effects the description of the initial state of the world. Links are important for encoding temporal information which is encoded in the *predecessor* and *successor* links between nodes, and by the use of split and join nodes.

Split and *join* nodes allow unordered, possibly parallel, actions. Split nodes have multiple successors, and join nodes have multiple predecessors, so that nonlinear plans can be produced. In fact, join nodes also have multiple successors, though the context selects only one of them to be in the current plan. The other successors represent the remainder of the plan after alternative operator expansions have been applied within the split-join pair.

SIPE assumes that whenever a split-join pair is introduced, whether by application of an operator or in the original goals, the main effects of the last node in each parallel branch are intended to be true at the point in time represented by the join node. This is enforced by placing a *parallel-postcondition* slot on each such join node which specifies the predicates that must all be true in the situation represented by the join node. This is done only when the join node is first introduced into the plan; it is not updated as more detailed levels of the hierarchical plan are expanded. As long as the highest level predicates are as desired, it is assumed that the lower-level predicates are irrelevant. If a join node originally has N predecessor branches, there will be N conjunctions of formulas that must all be true at the join node. (After planning, some branches may have been linearized, so there may be fewer than N predecessors.) An alternative way to encode this information is to have preceding nodes specify protect-until links that point to the join node. Parallel postconditions are preferred since they collect all this information at the join node itself, making it easier to reason about and reachieve these conditions during execution monitoring.

An example of the use of parallel postconditions is the standard block-world problem that has parallel goals of getting A on B and B on C. The join node at the end of these parallel goals has a parallel postcondition which requires both of these goals to be true at that point in time. Suppose the planner decides early in the planning process to linearize the parallel branches in order to get B on C before it attempts to get A on B, and then undoes the B-on-C property of the state while getting A on B. The parallel postcondition enables the planner to see that the plan so produced is invalid, since both of the goals are not true at the end of the plan.

SIPE implements conditional plans by providing *cond*, *endcond*, and *condpattern* nodes. The first two are similar to split and join nodes in their use of multiple predecessors and successors, but each successor of the cond node begins with a condpattern node that determines which successor will be executed. The first successor of a cond node whose condpattern node has a goal that is currently true will be executed. When branches are joined by an endcond node, the system assumes that the world is generally the same no matter which branch is taken. If this is not true, the user should not use the cond-endcond construct, but rather should produce alternate plans for each of the different worlds. To be more exact, "generally the same" means that if something is made true on any one branch of the cond-endcond, then the system will assume it is true after the endcond node. Thus, for the system to be consistent about the world after the endcond node, the branches in the cond-endcond must not change any aspects of the world upon which the remainder of the plan depends.

The system automatically inserts *choice* nodes in the plan to denote branching points in the search space. They have multiple successors, but the *context* selects one of these as being in the current plan. A choice node and one of its successors is referred to as a *choice point*. A context is a list of choice points, and uniquely determines a plan. Constraints on variables are posted relative to choice points. Thus, if the part of a plan after a choice node is removed, the corresponding choice point in the context should also be removed so that constraints that are no longer valid will be ignored. As we shall see, this capability is of critical importance in the replanner.

Goal nodes do not occur in final plans, since they represent problems that have not yet been solved. A goal node specifies a predicate that must be achieved, but which is not true in the situation represented by its location in the plan. Each goal node has a protect-until slot, which denotes that the goal must be maintained as true until the goal/node which is its

protect-until is achieved/executed.

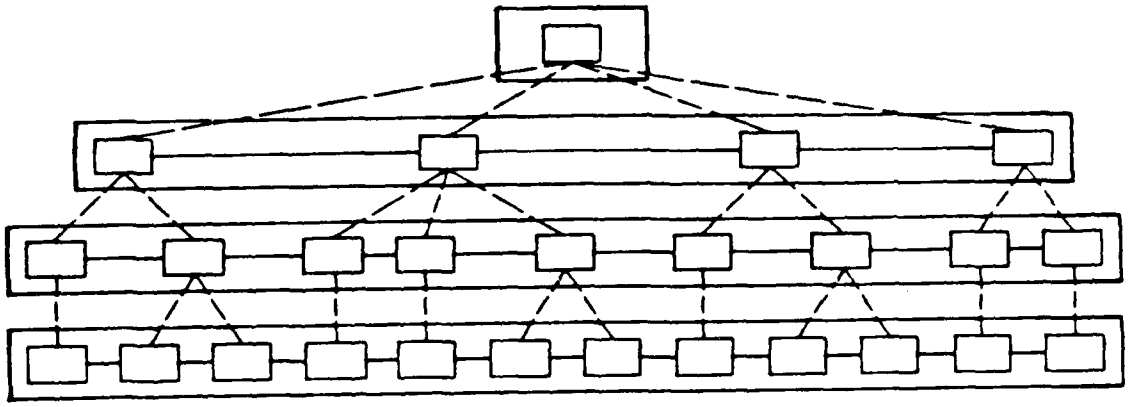
Phantom nodes are similar to goal nodes except that they are already true in the situation represented by their location in the procedural network. They are part of the plan because their truth must be monitored as the plan is being executed, and they may again become goal nodes.

Process and *choiceprocess* nodes represent actions to be performed during execution of the plan; they also have protect-until slots, as do phantom and goal nodes. In a final plan, all process nodes will denote primitive actions. As noted earlier, choiceprocess nodes specify which set of operators should be used to achieve the goal, and process nodes specify one particular action that must be used (and therefore do not produce a backtracking choice). Goal nodes implicitly state that any possible operator should be used to achieve the goal.

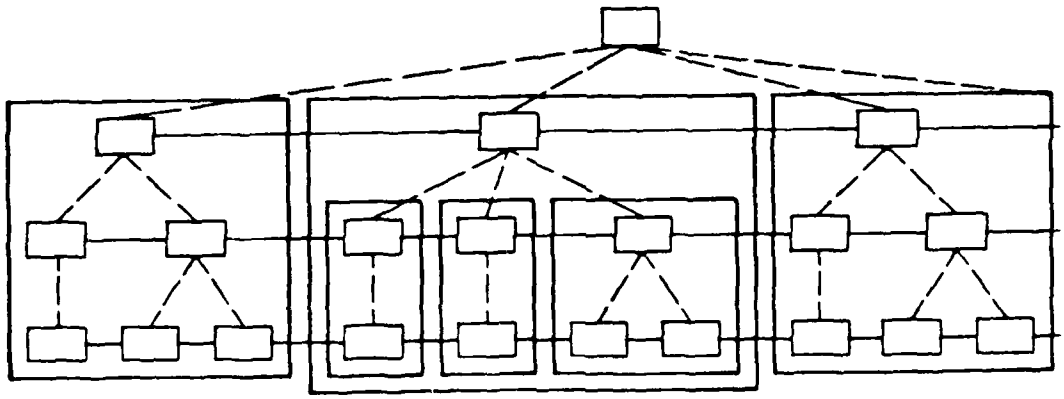
Precondition nodes provide a list of predicates that must be true in the situation represented by their location in the procedural network, as discussed earlier. They mean, in effect, that the part of the plan associated with them (see below) was produced on the assumption that the predicates in the precondition were true.

In addition to the horizontal protect-until, predecessor, and successor links within one level of a plan, there are vertical links between different levels of the hierarchy. Each node that is expanded by the application of an operator has descendant links to each node so produced. The descendant nodes in turn have ancestor links back to the original node one level higher in the hierarchy. Starting with a node that was expanded by an operator application, a *wedge* of the plan is determined by following all its descendant links (in the current context) repeatedly (i.e., including descendants of descendants, and so on) to the lowest level. (This definition of wedges is the same as that used by Sacerdoti [27].) Figure 3-4 depicts this graphically, with the large boxes in Part (b) representing wedges. The node originally expanded by an operator application is called the *top* of the wedge. A wedge with its top at a high level in the hierarchy will generally contain many lower-level wedges within itself. Only goal, process, and choiceprocess nodes can be the tops of wedges.

Note that precondition nodes can be used to delineate the subplan produced as a result of any one operator application at any hierarchical level in the plan. Since precondition nodes are created when an operator is applied and then copied down to lower planning levels, the part of a plan associated with them can be found by ascending along the ancestor links to the



(a) Plans at Different Levels



(b) Wedges Used by the Execution Monitor

Figure 3.4: SIPE Plan Viewed from Different Perspectives

point at which the precondition first became part of the plan. The node that was expanded by an operator to create this precondition is one level higher than where the precondition node first appears and is the top of the wedge associated the operator application.

Chapter 4

Constraints

One of SIPE's most important advances over previous domain-independent planning systems is its ability to construct partial descriptions of unspecified objects through the use of constraints. This ability is important both for domain representation and for finding solutions efficiently (since decisions can be delayed until partial descriptions provide more information). While constraints have been used in domain-specific planning systems [29], the constraints themselves have been domain-specific, making them less useful for solving different problems. SIPE is the first domain-independent planner to use domain-independent constraints.

Planning variables that do not yet have an instantiation can be partially described by setting constraints on the possible values an instantiation might take. This allows instantiation of the variable to be delayed until it is forced or until as much information as possible has been accumulated, thus preventing incorrect choices from being made. Constraints may place restrictions on the properties of an object (e.g., requiring certain attribute values for it in the sort hierarchy), and may also require that certain relationships exist between an object and other objects (e.g., predicates that must be satisfied in a certain world state). SIPE provides a general language for expressing these constraints on variable bindings so they can be encoded as part of the operator. During planning, the system also generates constraints that are based on interactions within a plan, propagates them to variables in related parts of the plan, and finds variable bindings that satisfy all constraints.

Constraints let both operators and plans be expressed more concisely and clearly. SIPE can declare a variable as `CARGOPLANE1 WITH RANGE 3000` which concisely represents the re-

quirements on the variable representing an airplane without adding any complexity to the plan. NOAH, for example, would have to represent every property of an object as a predicate and then achieve these predicates as goals during planning. In addition, constraints help the system represent plans that could not be expressed at all in previous systems. For example, the *optional-same* and *optional-not-same* constraints (described below) used in resource reasoning cannot be expressed as goals or preconditions in a system like NOAH.

Constraints also improve efficiency. Having a partial description of a planning variable allows large parts of the search space to be pruned, since only preconditions that are consistent with the partial description will be true. The constraint satisfaction algorithm also takes advantage of the sort hierarchy representation. For example, SIPE finds a cargoplane with the proper range in a single efficient lookup in the sort hierarchy. In NOAH, a cargoplane would be chosen without regard to its range, even though the range goal may later fail. If NOAH were to backtrack (in actuality it would fail), it would still have to search the whole space in the worst case.

4.1 SIPE's Constraint Language

Since no previous domain-independent planning systems have used constraints to partially describe objects, the constraints in SIPE will be documented in some detail. Their computational complexity will be mentioned in the following section and described in detail in later chapters. The system has two distinguished classes for variables representing numerical quantities, *numerical* and *continuous*. They are used below and described in Chapter 10.

The allowable constraints in SIPE on a variable V are listed below:

- **Class** This constrains V to be in a specific class in the sort hierarchy. In SIPE's operator description language, class constraints are generated implicitly based on the variable name. As we will see in the discussion of deductive causal theories, it can also be useful to provide explicit class constraints, e.g., `OBJECT1 OF TYPE BLOCKS`.

- **Not-Class** V must be instantiated so that it is not a member of a given class. For example, an operator could declare that an airplane be any airplane except a member of the class cargoplane with the following phrase: `PLANE1 IS NOT OF TYPE CARGOPLANE`.

- **Pred** — V must be instantiated so that a given predicate (in which V is an argument of the predicate) is true. This results in an explicit number of choices for V's instantiation, since all true facts are known (by the closed-world assumption). Pred constraints are generated automatically by the system during the planning process. They are the most frequently occurring constraint, and efficiently computing them is of central importance (see Chapter 5).

- **Not-Pred** — V must be instantiated so that a given predicate (in which V is an argument of the predicate) is not true. These are also generated by the system. Although not as ubiquitous as pred constraints, they are important.

- **Same** — V must be instantiated to the same object to which some other given variable is instantiated. This encodes a codesignation constraint between two variables, and is often generated automatically by the system.

- **Not-Same** — V must not be instantiated to the same object to which some other given variable is instantiated. In the block world Puton operator of Figure 3-3, the phrase `IS NOT BLOCK1` results in a not-same constraint being posted on both block1 and object1 that requires they not be instantiated to the same objects. Thus, if SIPE is looking for a place to put block A, it will not choose A as the place to put it (something that NOAH could have done in the block world, it seems).

- **Instan** — V must be instantiated to a given object. This could be represented by using a same constraint applied to objects as well as variables, but instantiation is a basic function of the system and warrants its own constraint for a slight gain in clarity and efficiency. These are generated automatically by the system as it fills out the plan.

- **Not-Instan** — V must not be instantiated to a given object. Constraints of this type can be generated by the system or provided as part of the domain description. For example, an operator can specify a variable to be any block except A by using the following phrase: `BLOCK1 IS NOT A`.

- **Optional-Same** This is similar to the same constraint, but merely specifies a preference and is not binding. For example, one would prefer to conserve resources by making two variables be the same object, but, if this is not possible, then different

objects are acceptable. These constraints are generated by the system as part of its resource reasoning.

- **Optional-Not-Same** — This is similar to not-same, but is not binding. If SIPE notices that a conflict will occur between two parallel actions if two variables are instantiated to the same object, then it will post an optional-not-same constraint on both variables. If it is possible to instantiate them differently, a conflict is avoided. If it is not, they may be made the same - but the system will have to resolve the ensuing conflict (perhaps by not doing things in parallel).

- **Any attribute name** — This requires a specific value for a specific attribute of an object. `BLOCK1 WITH COLOR GREEN` would constrain the variable to match only green blocks. Numeric values can also be compared with *greater than* and *less than*. In planning an airline schedule, for example, the operator used for cross-country flights might contain the following variable declaration: `PLANE1 WITH RANGE GREATER THAN 3000`. This would have created a constraint on plane1 requiring the range attribute (in the sort hierarchy) of any possible instantiation to have a value greater than 3000.

- **Current Value** — A numerical variable can be constrained to be the current value of a continuous variable at some point in the plan. This permits operators to reason about and place constraints on the value that some continuous variable has at some particular point in time. The syntax for specifying this in a SIPE operator is `NUMERICAL1 IS CURRENT VALUE OF CONTINUOUS1`.

- **Previous Value** — This is the same as current-value except that the value is taken just before the current node instead of just after it. The syntax is `NUMERICAL1 IS PREVIOUS VALUE OF CONTINUOUS1`.

- **Range** — A variable can be constrained to lie within a certain range e.g., `NUMERICAL1 IS [5,20]`.

- **Function** — A variable can be constrained to be the value of a certain function applied to any number of arguments. The function should return a numerical value when called with fully instantiated arguments. If some of the arguments are not instantiated, SIPE will compute a range from the function constraint by calling the function on all the possible instantiations. The ability to specify functions is of general utility.

- **Summary Range** — Since computing numerical constraints can be expensive, it was necessary to address the problem of choosing between storage and recomputation. Our solution is to store the results of computing a numerical variable's constraints by placing a summary-range constraint on the variable. (This cannot be done for continuous variables because their values vary with time.) These constraints are not posted by users, but only by the system, and are used during matching so that function, range, previous value, and current value constraints can be ignored. Summary ranges are recomputed at appropriate intervals.

4.2 Using Constraints

Much of SIPE's expressive power and efficiency is rooted in the ability to reason about constraints. Constraints add considerably to the complexity of the planner because they interact with all parts of the system. The most basic operation of the planner is the application of the truth criterion, the most basic operation of which is unifying two variables, which in turn involves determining whether the constraints on the variables are compatible. In a similar way, constraints also interact with the deductive causal theory. Constraints also affect plan critics, since determining if two concurrent actions interact may depend on whether their constraints are compatible. SIPE must also solve a general constraint-satisfaction problem with reasonable efficiency, although how to control the amount of processing spent on constraint satisfaction is an open and important question. SIPE currently uses a simple and straightforward constraint-satisfaction algorithm that is modular and replaceable.

Currently, SIPE runs the global constraint-satisfaction routine only once per planning level in the hierarchy. This can be easily changed in domains where better performance might be achieved by investing this effort more or less often, and can also be invoked interactively by the user. While planning within a planning level, the system makes localized computations to assure that constraints are likely to be satisfiable. It is an open question how "local" to keep these computations. Checking the complete global constraint network on each variable match is prohibitively expensive, so SIPE employs an algorithm to localize this computation while keeping it broad enough to rarely, if ever, permit a variable to match when it should not. In the domains implemented so far, there has been no problem with invalid matches. This algorithm and the tradeoffs involved are discussed below.

The system immediately propagates consequences of a new constraint as soon as it is posted. The small cost of so doing is more than offset by the immediate discovery of unsatisfiable constraints, which prunes the search space. When same and not-same constraints are added to a variable, similar constraints are immediately propagated to all other variables involved. (This may result in forced instantiations.) Whenever a nonnumerical constraint other than these two is added to a variable, the system verifies that at least one object satisfies all the constraints on this variable. This is efficient because pred constraints store explicit disjunctive lists of possible instantiations. If only one possible instantiation remains for a variable, that instantiation is made immediately; if no instantiations remain, the current search branch fails immediately.

The addition of numerical constraints is handled slightly differently. The consequences of these constraints are summarized in the summary-range constraint, which must be appropriately recomputed as planning progresses and more constraints are added and more instantiations made. The addition of numerical constraints triggers a recomputation of the summary-range constraints of the variables, as described in Chapter 10. After instantiating variables, the recomputation problem is harder. The system keeps track of variables upon which a summary range depends and recomputes the range when any of these variables are instantiated.

Pred and not-pred constraints are posted by the system to ensure that preconditions of domain rules will be true. They are of central importance in the planning process, and constrain the instantiation of many different variables, thus requiring the previously mentioned localization of computation during variable matching. By accumulating them, the system is assured that any object satisfying all the constraints on a variable will have all the properties required by the plan. In Chapman's terminology, a pred constraint describes all the possible *establishers* and *white knights* of a predicate (see Chapter 5). A pred constraint specifies a set in which each element is a list of possible codesignations for variables that will assure that the predicate is made true. For example, to ensure that $P(x\ y\ z)$ holds, a pred constraint will be posted on each of the three variables. In general, the constraint specifies a set with elements $(x_1\ y_1\ z_1)$ through $(x_n\ y_n\ z_n)$. To satisfy the constraint, there must be some i such that the three variables $(x\ y\ z)$ can codesignate with $(x_i\ y_i\ z_i)$, respectively. (Two variables can codesignate if they unify, i.e., if it is consistent to assume they have the same instantiation.) Similarly a not-pred constraint specifies a conjunctive set where each element is a list

of possible codesignations for variables which must not all be present or the predicate will be made false.

It is fairly easy to write operators that will produce constraints that are computationally too expensive. The user must therefore be careful to formulate his domain in such a way as to ensure that this computational cost will be reasonable. For example, in the mobile robot domain, an initial attempt at a solution deduced something about every location that was not adjacent to some new location. This works fine when the location being moved to is instantiated; when it is not, however, the number of things that can be "not adjacent" to it is enormous. This causes a pred constraint with a huge number of disjuncts to be added to a variable, and this constraint must later be processed frequently during subsequent unifications. However, a second attempt at writing such a deductive rule produced efficient constraints and made use of universal quantification and the not-same constraint.

4.3 Unification

System efficiency depends upon the algorithm for unifying two variables with constraints. This algorithm underwent several changes during early development of the system before providing its current levels of acceptable performance. One of these changes included checking for and ignoring pred constraints that are subsumed by other pred constraints. While this does not affect the computation of the correct answer, it significantly improved system performance. Here we describe our solution to the central problem of handling pred constraints.

During the unification of two variables, pred constraints force the system to determine whether a set of instantiations will be compatible after these two variables are unified. In the above example, if the system is trying to unify the variable x with the object A, it must find an $(x_i\ y_i\ z_i)$ in the pred constraint on x such that x_i and A can codesignate. It should then check that the other members of this set (y_i and z_i in this case) can codesignate with their corresponding variables. Checking every member of the set in this way would propagate unifications throughout the system as variables are matched recursively. In the worst case, every constraint in the system might be checked each time a variable is unified. This is computationally unacceptable. Furthermore, loops may be created because the system may recursively unify the two original variables again.

To avoid the above problems, the unification algorithm does a complete check of constraints only at the top level of recursive calls. At lower levels of the recursion, the algorithm still checks all constraints, but this time recurses on pred constraints only for variables that are instantiated (effectively assuming the uninstantiated variables will be acceptable). This approach avoids both the problem of needing to check for loops and of the unification becoming prohibitively expensive. This algorithm has proven satisfactory in practice. It matches exactly the variables one would intuitively expect from looking at two levels of the constraint network. The idea of checking the instantiated variables at the lower levels of recursion is critical for achieving our level of performance, because it is frequently the case that most variables in a problem are instantiated. Although no invalid unifications have been detected in practice, the consequences of an invalid one would generally be the unnecessary searching of a portion of the search space that did not contain a solution (as the problem will be discovered at the end of each planning level when the global constraint satisfaction problem is solved). This would not be a catastrophe, although earlier versions of the matching algorithm that did permit invalid unifications caused problems in the replanning algorithm.

SIPE's unification algorithm is similar in spirit to that used by Allen for maintaining temporal relations [2]. Allen's algorithm guarantees only consistency between three-node subnetworks of the overall constraint network. The algorithm therefore permits inconsistent labelings of a network, but Allen still considers it the best practical solution, since it avoids an exponential search while producing useful results. In SIPE, the plan critics will eventually find any inconsistencies and initiate backtracking so that a valid solution will be found.

Not-pred constraints cause little problem. They specify one set of codesignation constraints that must not all simultaneously be true. This is used to avoid selecting a set of instantiations that will have the effect of some previous action making a predicate we care about false. In Chapman's terminology, not-pred constraints denote a *clobberer* of a predicate the system cares about. The local check made in unification is fast: if all these codesignation constraints must currently be true through direct means (e.g., same constraints between the two variables, or their instantiation to the same object), then the match fails, otherwise it succeeds. The case where the variables do not directly codesignate, but may do so implicitly by weight of the global body of constraints, is recognized when the global constraint satisfaction problem is solved at each planning level. This global satisfaction will fail if instantiations cannot be found which satisfy the not-pred constraint, and the system will then backtrack.

Chapter 5

The Truth Criterion

The truth criterion determines whether a given formula is true at a given point in time. Its efficiency is crucial to any quest for heuristic adequacy, as it represents the most basic operation of a planner. It, in turn, relies heavily on unification, which therefore must also be efficient. Chapter 4 described how SIPE's unification algorithm efficiently solves the problem posed by unification when constraints are present. Because of the central importance of the truth criterion, SIPE's is explained in detail in this chapter. Chapman [4] says (justifiably) that "SIPE's treatment of [derived effects] is incomplete and not generally correct". This chapter provides enough information to add precision to this statement. In particular, it will be shown exactly where the heuristics in SIPE cut corners, how the planner will later correct problems that are introduced, and why the truth criterion is useful in practice. (This is the most technical chapter in this report, and some readers may want to skip the latter parts of it.)

The efficiency of our frame problem solution comes from the assumption that no predicate changes its truth value except when it is mentioned in the effects of an action, and from special heuristics for avoiding the NP-complete problem introduced by parallel actions. These heuristics prevent SIPE from reasoning about all possible shuffles of the parallel actions. The former assumption is an extension of the STRIPS assumption, used in such seminal planners as STRIPS, NOAH, and NONLIN. The (strict) STRIPS assumption is that no predicate will change its truth value when an event takes place unless the event explicitly lists that predicate on its add or delete lists. SIPE extends this in two ways. First, it permits universally quantified variables in its effects, enabling one predicate instance to encode a whole set of

ground predicate instances. Second, there are no add and delete lists; all effects are in one list, which can contain both negated and unnegated predicates; and, most importantly, these effects can be deduced in a context-dependent manner rather than being specified in an operator.

In past planners, use of the STRIPS assumption has made operators unacceptably difficult to describe (see Chapter 6). SIPE avoids this by using constraints, resources, and deductive causal theories to reduce the representational burden of operators. The chapters on each of these topics describe how these mechanisms can be used to represent powerful operators, without explicitly representing all their effects. Since all deduced effects of an action are added to the action's effect list when the action is inserted in the plan, the truth criterion can operate solely on the effects of actions in the plan (similar to the way STRIPS uses the add and delete list of actions). In fact, in the STRIPS domain of ground, linear plans, the truth criterion is basically the STRIPS algorithm.

However, other features of SIPE, particularly constraints, variables, quantifiers, and non-linearity, complicate the problem. Because of the power of a constraint-based representation with variables, it is not enough for the truth criterion to return true or false. It will do so when such a truth value is forced upon a formula without further instantiations or posting of constraints, but in all other cases it should return a set of constraints that can be posted to make the formula true. The truth criterion returns such sets composed of same, not-same, instan, not-instan, pred, and not-pred constraints. Unlike other classical planners, limited forms of disjunction and existential and universal quantifiers can occur in certain places in the system. As we shall see, details of the system's representation determine restrictions on quantifiers and disjunction. SIPE's quantifiers are restricted versions of the quantifiers in first-order logic, but are nevertheless quite useful.

Another slight complication is that all predicates in SIPE can have three values as their sign -- they can be negated, unnegated, or unknown. Like many previous planners, SIPE makes a closed-world assumption, assuming that if a predicate is not given in the world model, then its negation is true. (This means that the user does not have to axiomatize the enormous number of formulas that are not true in his domain.) This is not critical; the system could easily be changed to assume that a predicate's truth-value is unknown unless an explicit mention of the predicate is found in either negated or unnegated form. The use of a deductive causal theory does not violate the closed-world assumption; it is used only to

deduce effects of an action when the action is added to a plan (thus sparing the operator that represents the action from having to specify these effects).

5.1 The Formula Truth Criterion

SIPE's formula truth criterion (FTC) determines the truth of a formula at a point in time. The latter is represented by a particular node in a plan which we will call the *current node*. Since a formula is a conjunction where each conjunct is a predicate or a limited form of disjunction, the FTC is easily reduced to a predicate truth criterion (PTC). This reduction is done by calling the PTC in turn for each conjunct in the formula. If a conjunct is of the form *(OR pred1 pred2 . . . predN)*, the semantics are that the system will use only the first predicate that has possible matches. (Thus, if pred1 has some possible matches, they are the only matches that will ever be considered.) This rendering of disjunction has proven useful in practice without introducing additional complexity, and its use is restricted — for example, disjunctions may appear in preconditions of operators but not in effects of actions.

When the FTC calls the PTC and receives a set of constraints in return, these constraints are immediately posted in such a way that they (and any propagated consequences) can be deleted should a later predicate in the formula fail to be satisfied. This posting of intermediate constraints is a vitally important part of the FTC, because these constraints often greatly reduce the number of possibilities to be considered during later calls to the PTC while matching the same formula. This is somewhat complicated — in addition to the possible deletion of consequences already mentioned, the most concise constraint cannot be formulated until all predicates have been matched. Thus, the constraints added earlier in the formula might be large and could slow down all future matching. SIPE solves this problem by implementing a check for constraint subsumption.

To explain the PTC, we will first describe the algorithm for ground, nonlinear plans, i.e., plans with only ground instances of predicates as effects (this is the simple context in which STRIPS operated). Then we will introduce the enhancements to this algorithm necessitated by variables, quantifiers, and nonlinear actions. The PTC also has special techniques for the computation of numerical quantities, but these are described in Chapter 10 with the other algorithms for numerical reasoning.

To determine the truth of a ground predicate (referred to as the *query* predicate) in a ground, linear plan, the truth criterion simply looks backward from the current node until it finds an effect that has the same arguments as the query predicate (regardless of the sign of the effect). If the sign of the effect (which can have three values) is the same as that of the query predicate, then the query is true, otherwise it is false. If the PTC retreats to the planhead node of the plan without finding the query predicate as an effect, then the predicate is true if it is negated, and false if it is either unnegated or unknown. Thus, a predicate that has *unknown* as its sign will only be found to be true if there is an effect denoting that the predicate is unknown. Any predicate that is true by virtue of being true initially and remaining unchanged will be matched just as would a predicate that was made true by a later action, since the description of the initial world is the effects of the planhead node at the beginning of the plan.

5.2 Introducing Variables

Once variables are permitted in predicates, the PTC becomes more complex. Effects no longer determine the truth of a predicate, rather they are possible matches for the predicate, depending upon how the variables are instantiated. We will describe the complications introduced by this, without initially considering additional complexities imposed by quantifiers. SIPE variables differ from variables in logic. The former are introduced into the plan by application of an operator which may place constraints on a variable before it is introduced. The argument slot of an operator can specify constraints, and constraints are added by the truth criterion when matching the operator's precondition (so that a variable will match all and only those objects for which the precondition is true). Thus a variable is not a true wild card, but rather something that matches only certain objects as determined by its constraints.

First, let's define some terminology. Chapman [4] introduces the notions of *establisher*, *clobberer*, and *white knight*. An establisher is an effect which necessarily asserts a predicate (in the ground case, all assertions were necessarily the case), a clobberer is an effect that possibly denies a predicate (depending on variable instantiations), and a white knight is an effect that will reassert the predicate denied by a clobberer in any situation where variable instantiations actually make the latter deny the predicate. Two variables/objects *necessarily codesignate* if they are the same objects, are instantiated to the same objects, or are constrained to

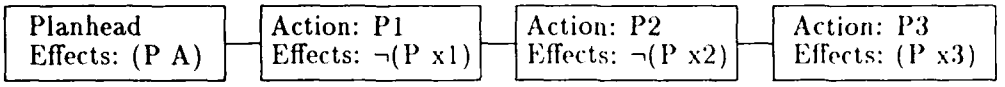


Figure 5.1: Effects with Variables in a Generic Plan

be instantiated to the same object (with the same constraint). Two variables can *possibly codesignate* if they unify, i.e., if it is consistent to assume they have the same instantiation. We will say that two instances of the same predicate necessarily/possibly codesignate when the corresponding variables in the arguments of each predicate (except for numerical variables) necessarily/possibly codesignate. Whenever the term *codesignate* is used without a modifier, it means that the objects in question *necessarily* codesignate. Possible codesignation will always be referred to explicitly.

An example will help to explain these terms and will be useful for explaining the PTC. Suppose we are trying to determine the truth of $(P\ A)$ at the end of the plan in Figure 5-1, where A is an object and x1, x2, and x3 are variables. In this case, with Chapman's terminology, the effect of the planhead node is an establisher, the effects of P1 and P2 are clobberers, and P3 could be a white knight for either P1 or P2. It would be a white knight for P2 if the system enforced a codesignation implication saying that whenever x2 designated A then x3 also designated A. SIPE does not use the notion of a white knight because specifying such implications can be too complex and inefficient. As Chapman says, "making [a codesignation implication] is tricky; this can not directly be expressed as a constraint" [4]. One way of enforcing this implication would be to make x2 and x3 codesignate, but this makes a commitment that may prevent a valid plan from being constructed.

Even if there were a way to express this constraint efficiently, there is the problem of choosing for which clobberer to be a white knight. We assume the actions were not inserted in the plans specifically to be a white knight — for example the plan in Figure 1 might have been constructed to solve some goal other than P (not all the effects are shown), and when the planner later discovers that it needs $(P\ A)$ to be true, it will have the problem of assigning white knights to clobberers to make it true. If there are n possible clobberers followed by n possible white knights, then there are n^2 ways to match them one-to-one, but the problem is not so simple as one-to-one matching. In fact, there is a combinatorial explosion — one

white knight might take care of a whole set of codesignating clobberers, and, as new actions and constraints are generated, it may be best to reassign white knights to clobberers.

SIPE's solution to this problem does not distinguish between white knights and establishers. Roughly, both are treated as "possible establishers" which we will refer to as *p-establishers*. The problem is solved by posting pred and not-pred constraints to assure a p-establisher that is not clobbered, given the current state of the plan and codesignation constraints. This is done without making any commitments that are not forced, thus introducing no branch points in the search. The disadvantage of this scheme is that these constraints need to be recomputed when new codesignation constraints are added. Since only constraints compatible with the existing ones are added, recomputation represents more a narrowing of possibilities than a jump to a different solution.

There is a tradeoff in determining how often to recompute such constraints. Recomputing them all whenever a new codesignation constraint is added is obviously inefficient. Instead, SIPE recomputes them for each variable in a predicate every time that predicate is checked with the truth criterion. There is almost no cost involved, since the truth criterion must essentially compute these constraints in any case. Thus, this recomputation of pred and not-pred constraints is preferred to specifying codesignation implications, both for efficiency and lack of commitment — note that the system essentially "reassigns" white knights dynamically as the plan evolves. The only theoretical problem with this scheme is proving that the recomputations happen frequently enough. SIPE attempts to ensure this; for example, all preconditions of operators that have been applied are recorded in the plan, and every time a new operator is applied the system checks all preconditions that come after it in the plan. This assures that no addition to the plan will negate assumptions on which the remainder of the plan is based. These mechanisms should recompute the pred and not-pred constraints on all variables used in goals or preconditions before the constraints are used by the other critics. One could not prove that SIPE always recomputes these constraints often enough — certainly under interactive control, the user can allocate resources whenever he wishes. However, this has never been a problem in use of the system, and the automatic search could easily provide such assurances at some computational cost.

To determine the truth of a query predicate with variables in a linear plan (ignoring quantifiers), the PTC looks backward from the current node, continually finding effects that possibly codesignate with the query predicate (regardless of the sign of the effect). It main-

tains a list of p-establishers and clobberers for this query, adding effects that are possible matches to these lists when appropriate. If the sign of the effect and the sign of the query predicate are the same, then the effect is a candidate p-establisher, otherwise it is a candidate clobberer. However, these candidates are never added to the lists if they necessarily codesignate with any effect already on either list. Thus if a clobberer is followed in the plan by a p-establisher that codesignates with the clobberer, the latter never becomes a member of the clobberer list. Likewise, a p-establisher followed by a codesignating clobberer never makes the p-establisher list.

This traversal of the plan can end in two ways. It ends when the beginning of the plan is reached, in which case the closed-world assumption will act as a codesignating establisher for negated query predicates and as a codesignating clobberer for all other predicates. It also ends whenever some effect necessarily codesignates (as either a clobberer or establisher) with the query predicate. The PTC thus produces a clobberer list, a p-establisher list, and a flag denoting whether termination of traversal was caused by a necessarily codesignating clobberer or establisher.

If the query predicate is constrained to codesignate with some member of the p-establisher list and to not codesignate with any member of the clobberer list, then it is necessarily true in the current plan, as so constrained. This is accomplished by posting pred and not-pred constraints (see Chapter 4). Sometimes the PTC is called simply to test applicability of an operator without actually applying it, in which case constraints do not get posted. In general, the PTC will convert the p-establisher list into a pred constraint, and the clobberer list into a not-pred constraint. However, when traversal ends with a codesignating establisher, then only the not-pred constraint is posted, since the query predicate will necessarily be true as long as none of the possible clobberers codesignates. Pred and not-pred constraints may be converted to same and not-same constraints when there is only one possible way to make a query true. When the PTC is again applied to a predicate codesignating with the query predicate, pred and not-pred constraints are recalculated by the new application of the PTC.

The rest of this chapter explains enhancements of the predicate truth criterion in great detail. The casual reader may want to skip ahead to Chapter 6.

State-rule: Deduce-clear
Arguments: object1,object2, object3 Class Existential;
Trigger: ¬(On object1 object2);
Condition: ¬(On object3 object2);
Effects: (Clear object2);

Figure 5.2: A SIPE Block-World Deductive Operator

5.3 Introducing Existential Quantifiers

Unlike other classical planners, SIPE permits limited forms of quantifiers. While these are restricted versions of the quantifiers in first-order logic, they are nevertheless quite useful. The closed-world assumption leads to restrictions on the use of quantifiers. Formulas of the form $\exists x.P(x)$ and $\forall x.\neg P(x)$ are relatively easy to compute, since they involve searching for occurrences of unnegated predicates. In SIPE, existential variables can appear only in preconditions (and *conditions*, as we shall see later), and are used to represent both these forms. Disjunction and existential quantifiers are not permitted in the world model or in the effects of actions -- otherwise, the truth of a predicate could not be determined directly since it would depend on which disjunct was true or to which [unknown] object the existential variable referred.

In negated predicates, the existential quantifier is scoped (by definition) within the negation, yielding a formula similar to the previous universally quantified formula. While similar, these formulas are not equivalent because of restrictions (described below) that are placed on the meaning of existential quantifiers in order to compute them efficiently. The scope of each existential quantifier is local to the predicate in which it occurs. Let us consider the block-world Deduce-clear deductive operator in Figure 5-2. The use of such operators will be explained in Chapter 6; here we are concerned with the matching of its condition by the PTC. Deduce-clear is used to deduce that a block is clear in any situation, even in a world where several blocks may be on top of another block. In this example, Object3 is declared to be an existential variable, with the quantifier scoped locally to the predicate in the condition, within the negation. Thus, the PTC is trying to determine the truth of a formula similar to the first-order logic formula $\forall object3.\neg On(object3\ object2)$. Deduce-clear will deduce that a block originally supporting N blocks will be clear only after N actions remove blocks.

For an unnegated query predicate with an existential variable as an argument, the only change to the PTC makes it more efficient. In the case of unary predicates, any p-establisher becomes a codesignating establisher that terminates the traversal of the plan (since all we care about is whether or not some binding exists for the variable). For predicates with more arguments, the same termination results when all nonexistential arguments necessarily codesignate (assuming all existential arguments possibly codesignate).

Determining the truth of a negated query predicate with an existential variable, such as that in Deduce-clear, is more complex. For each predicate occurrence that is a possible clobberer of the query predicate, the system must find a p-establisher that follows this clobberer in the plan. To accomplish this, the PTC constructs a list of codesignation constraints that it will temporarily assume for the duration of the match of this query predicate. These temporary constraints match each clobberer to a p-establisher to determine whether there is some way, given the current plan and constraints, to make the predicate true. The temporary constraints are never posted, and do not become part of the pred or not-pred constraints that are generated (any constraints that would be posted without the existential are still posted). Thus no constraints force the system to maintain the truth of the query predicate. As with pred and not-pred constraints, SIPE relies on the fact that the PTC is recomputed every time the truth of the predicate is checked.

The combinatorics of possible ways to match clobberers to p-establishers are eliminated by assuming that each new clobberer the PTC adds to its list must match a different p-establisher, and that the PTC simply matches it to the first p-establisher that will work. Thus there is no search involved, but we only compute the true first-order logic equivalent of the quantified formula in cases where variables that do not necessarily codesignate can be assumed not to codesignate. Note that if two clobberers necessarily codesignate, then the second one is never added to the PTC's list and is therefore (implicitly) matched to the same p-establisher as the first one. This algorithm maintains the "least commitment" approach by not assuming codesignation constraints until forced to do so. While not providing a complete existential quantifier (which cannot be done without sacrificing heuristic adequacy), SIPE does provide a useful tool that is certainly better than not having anything, which is the choice in other classical planning systems. The user can employ this tool when his problem fits within its restrictions, and ignore it in other cases.

The existential quantifier provided by SIPE is useful for three reasons — it has proved

useful in practice; many domains do have the property that new variables are generated only when they do not codesignate with existing variables; and it is still useful in domains where codesignations change. One of the reasons for the latter is that the PTC is constantly being recomputed for this predicate and will react to the addition of new codesignation constraints. Furthermore, the only effect of not forcing codesignations in order to match an existential variable is than some operators may fail to be applicable when one might wish them to be — the system does not produce invalid plans because of it. This can be cured simply by providing more operators to the system.

The Deduce-clear operator provides an example of the usefulness of such a tool. It can deduce that a block originally supporting N blocks will be clear only after N actions remove blocks. This will succeed even if the N actions that remove blocks have not yet been instantiated (which means that it is not yet known which blocks will be moved). The restrictions on existential quantifiers cause no problem in the block world because different actions that put noncodesignating blocks (variables) on another block can end up with these blocks codesignating only if there is an intervening pickup action between the puton actions (in which case there is still a p-establisher for each clobberer using SIPE's algorithm). One would expect other interesting domains to have this property.

5.4 Introducing Universal Quantifiers

SIPE's universal quantifiers differ from their counterpart in logic, but again provide a useful capability. Universal variables are permitted in the effects of actions, and are inserted primarily by the deductive causal theory. As expected, a universal variable in the effects of a node means that the effects are true for all objects that match the variable (taking into account the constraints on the variable). However, universal variables in preconditions are treated like any other variable (i.e., the quantifier is ignored) — the only use of quantifiers in preconditions permitted by SIPE is implemented through existential quantifiers (which can implicitly do a limited form of universal quantification on negated predicates). Thus, if X is a variable denoted as universal in P, a precondition, it means that only instantiations of X for which P is true will be considered hereafter (i.e., $x \mid P(x)$), not that P must be true for all possibilities (i.e., $\forall x.P(x)$).

When variables are introduced during operator application they are constrained by the

Causal-Rule: No-longer-nextto
Arguments: robot1, location1,object2 Class Universal;
Trigger: (At roboti location1);
Precondition: (Nextto robot1 object2);
Condition: ¬(Adjacent-loc object2 location1);
Effects: ¬(Nextto robot1 object2);

Figure 5.3: Causal Rule for Updating Nextto in the Robot Domain

system to match all and only those objects for which the precondition of the operator is true (as well as inherited constraints specified in the argument list of the operator). Combining such a constrained variable with SIPE's universal quantifier effectively represents a certain subset of objects that can be used in the effects of actions (i.e., an effect predicate with a universal variable can be read as "for all objects that unify with the universal variable, this predicate is true").[1] This ability to form subsets is useful and powerful, and exploits the system's representation and algorithms for the purpose of representing a number of assertions compactly as a single predicate with a universal variable. A primary advantage of this approach is the gain in efficiency that is achieved by matching only one predicate when calls to the PTC regress back to the node containing the universal variable in its effects, instead of having to match a predicate once for each member of the subset represented by the universal variable.

The mobile-robot domain provides an example of the use of universals. In one solution, we keep track of all objects that are next to the robot. The causal rule shown in Figure 5-3 adds $\neg Nextto$ predicates to the effects of any action that moves the robot. (This effectively eliminates the *Nextto* predicates that are no longer true after the robot has moved.) When the precondition predicate (*Nextto robot*1 *object*2) is matched, it will constrain the variable *object*2 to match only those objects that were next to the robot before it moved. The condition predicate will further constrain *object*2 to not be any object that is adjacent to the new location of the robot. Thus the constraints on the universal *object*2 ensure it will match exactly those objects that the robot was next to before it moved but was not next to

[1] Note that not all arguments of a deductive rule are universally quantified in this way. Some are parameters of the action for which we are deducing effects, while universal variables do not depend on the parameters of the action.

afterwards. This effectively picks out the subset of objects that interests us and allows their efficient representation.

Note that the constraints on *object*2 may refer to variables instead of actual objects. For example, the constraints may specify that *object*2 must match one of a set of N planning variables. Some of these variables may not yet be instantiated, but eventually they will be. Perhaps they will be instantiated to N different objects, or they may all be instantiated to the same object. There is no matching problem because the constraints on each of the N planning variables specify all relevant information (e.g., which variables and objects these N planning variables are or are not identical to).

The further planning of actions occurring either earlier or later will not affect the validity of the universal variable in the *Nextto* predicate that occurs in the deduced effects. This is true by virtue of the place where the predicate is recorded in the plan, because the truth criterion regresses back through a plan searching for effects that can match a given formula. Suppose N1 is one of the N planning variables that will match the universal variable. Let us consider the two cases of effects that occur after or before the *Nextto* predicate. If a later action specifies (*Nextto Flakey N*1) as an effect, this latter predicate will always be matched to a query before the predicate with the universal variable. Thus, the appropriate relationship between N1 and the corresponding variable in the query will already be determined and will not be affected by any subsequent attempt to match the query with the predicate containing the universal variable. Now, let us consider the case where further planning of an action before the one containing the *Nextto* predicate as an effect will be done at the next lower [hierarchical] planning level. In this case, SIPE recomputes all deductions that follow at that lower planning level. In particular, it deduces a *Nextto* predicate at the new level whose variables and constraints have been properly calculated for the new situation.

Changes to the PTC required by universal variables are detailed and not of great importance. Primarily they involve the fact that matches to universal variables can often be considering necessarily codesignating. In early stages of its development, SIPE did not permit a variable marked universal to appear in a precondition or have additional constraints. Allowing them to have constraints made them much more useful but entailed a major change, because before a universal variable always matched as necessarily codesignating. With constraints, universals are sometimes only possibly codesignating. For example, when adding to its clobberer or p-establisher lists, the PTC must add a predicate with a universal variable,

even if it matches an earlier entry that does not contain a universal variable (thus treating the match to the earlier entry as noncodesignating). Similarly, the PTC's traversal of the plan cannot be halted by matching the query predicate with an effect predicate containing a universal predicate.

The PTC's traversal of the plan is halted in one special case. Certain predicates are "unique" in the sense that they have one argument with the property that no other instances of this predicate can be true if they vary only this one argument. For example, a block can be on only one other block. The effects of creating such a situation can be easily expressed by saying the unique predicate is true and that the same predicate is not true when the unique-argument is replaced by a universally quantified variable. For example, in SIPE one can have the effects (*On block*1 *object*1) and ¬(*On block*1 *object*2) where object2 is universally quantified and constrained not to be the same as object1. SIPE halts the PTC's plan traversal when a pair of such effects assure that all possible matches have been considered. Briefly, this happens when two effect predicates like those above necessarily codesignate with the query predicate in all argument positions except one. When the two predicates have opposite sign, their two entries in this argument position have a not-same constraint each referring to the other, and one of the arguments is universally quantified, then the traversal of the plan can be terminated because all possibilities have been covered.

While the not-same universal special case may seem trivial, it has proven quite useful in practice (see Chapter 6). While the effect of terminating the PTC early is desirable, the primary advantage comes from the generation of much smaller pred and not-pred constraints, especially in long plans. This makes unification of variables in every part of the system (e.g., the plan critics) faster. Such algorithms are discovered only by implementing and using a planner, and analyzing how it can be made more efficient.

5.5 Introducing Nonlinearity

As Chapman has shown, nonlinearity makes the truth criterion NP-complete (given a reasonably powerful representation). SIPE allows a restricted nonlinearity and provides mechanisms for circumventing the NP-complete problem within this nonlinearity. One of the basic restrictions is that given a set of parallel subplans, SIPE will only reorder them by putting a whole subplan before or after the others. Thus it can not produce all possible shuffles of

the primitive actions in the subplans. This reduces the number of possibilities enormously. While this restriction is not acceptable in complex scheduling problems, it is quite useful in problem-solving types of tasks. Classical planners are not good tools for big scheduling problems in any case - - they would be better as part of a larger system in which the classical AI planner produces a plan that meets certain goals and then gives this plan as input to a scheduling system that applies powerful computational techniques.

An important technique for dealing with nonlinear actions is distinguishing between main effects and side effects of an action. The system guarantees that the main effects of parallel actions will all be true at the end of the nonlinear part of the plan. This is implemented by the plan critics which are described in Chapter 9. While matches to these main effects are guaranteed to be correct, for other predicates the PTC merely proves that there is one possible ordering of the parallel actions that makes the predicate true without enforcing that order.

Proving that there is one possible ordering is very efficient - the ordering itself need not even be calculated. When the PTC regresses to a join node that marks the beginning of a set of parallel branches, it recurses on itself for each branch, constructing new lists of *p-establishers and clobberers* for each branch. The recursively entered PTC has copies of the top level versions of these lists (as they existed at the join node), and effect predicates are not added to the new lists unless they would also have been added to the top level versions. They are added to the new lists just as the would have been to the top level lists. Termination of the traversal of the plan is also done for the same reasons. Necessarily codesignating effect predicates terminate the traversal of the current parallel branch when they make the query predicate false, and of all parallel branches when they make the query true. Not-same universals and disproofs of negated existentials (as discussed above) can also terminate traversal of all parallel branches.

Once the PTC has collected the new list of p-establishers and clobberers for each parallel branch, it appends all the p-establishers and adds them to the top level p-establishers list using the normal mechanism (which eliminates duplicates). After this, all the new clobberers are added to the top-level clobberer list using the normal mechanism. Thus, a clobberer will not be added (to the top-level list) if there is a p-establisher on any parallel branch that codesignates with it. Thus, the algorithm efficiently calculates whether there is any possible shuffle of the branches which will make the query predicate true. The top-level lists

so constructed can be used for continued traversal of the plan preceding the split node at the beginning of the parallel branches.

The PTC treats conditional branches in conditional plans in the same way as parallel branches. The system effectively assumes that if something is made true on any conditional branch then it is true after the endcond node. This implements the assumption that the world is pretty much the same no matter which branch is taken (as discussed in Chapter 3).

Because the PTC only finds that some ordering makes a query predicate true without calculating what the order actually is or enforcing it, different calls to the PTC could return values based on contradictory orderings. Thus invalid plans might temporarily be produced (though only in nonlinear cases). However, there are mechanisms for preventing the system from making contradictory assumptions about different orderings as planning proceeds. For example, once an operator is applied based on the truth of its precondition (which may implicitly assume additional partial order on earlier parallel actions), this precondition is recorded as being true at that point in time so that no later action can assume otherwise. Similarly, goal and phantom nodes record the truth of their effects for all succeeding nodes. Because of this, different calls to the PTC for the same predicate could not make contradictory ordering assumptions because earlier assumptions would effectively be encoded in precondition, goal and phantom nodes. Thus, contradictory ordering assumptions can only occur when the PTC finds values for two different predicates that are both side effects in the unordered subplan, and it happens that no one ordering can make both values true. The discussion below explains that such contradictions are often only temporary, and that there are ways to avoid them. (One somewhat far-fetched advantage of this is that the two predicates could both be true in some shuffle that SIPE itself could not generate, but that could be generated by a specialized scheduling algorithm that took the SIPE plan as input and considered the truth of various predicates as constraints on the ordering of actions.)

The above solution again follows the "least commitment" strategy of not committing to an order until it is forced. It has proved to be a useful compromise that provides the user with a powerful tool to produce useful plans efficiently. The potential for contradictory ordering assumptions has not been a problem in practice. The user can always encode all important predicates as main effects and the plan critics will then assure a correct plan. Since SIPE provides flexibility in specifying main and side effects, it is easy to change any problematic predicate into a main effect. Another reason for the success of this algorithm is the fact

that any contradictory ordering assumptions produced are often only temporary. Plan critics sometimes correct them, and further planning is often forced to make commitments about ordering. SIPE immediately reacts to further ordering constraints by updating the remainder of the plan to be consistent with the new ordering. This involves redoing all the deductions that come after the reordered part of the plan, as well as checking preconditions and phantoms. Since the implicit ordering assumptions are calculated for each predicate every time the PTC is called on the predicate, the system will immediately purge itself of any implicit contradictory ordering assumptions that are corrected by the addition of ordering constraints. Lastly, the system could make final checks to assure correctness at some computational expense. It is easy to identify predicates that may make implicit ordering assumptions. For each of these, the system could calculate (at linear cost) two subsets of parallel branches, one of which had to precede the other to make the predicate true. Then the requirements of all predicates could be checked for contradictions.

5.6 Summary

The truth criterion has proved to be a useful compromise that provides the user with a powerful tool to produce useful plans efficiently. This chapter describes all the heuristics that have been incorporated to provide this performance. Universal quantifiers improve system efficiency, and existential quantifiers, though not as powerful as their true first-order logic equivalent, provide the user with a new and efficient tool. Chapter 10 describes further facets of the truth criterion having to do with numerical quantities.

While it is easy to criticize the nonlinear algorithm because of the contradictory ordering assumptions that may be made, one should consider the alternative: solving an NP-complete problem. If one can afford to wait an arbitrary amount of time for a planner to determine if a particular predicate is already true or not, then Chapman and Pednault [22] provide sound algorithms. However, if one wants to use a planner to solve real problems, the algorithm above, or at least the tradeoffs it makes, will likely be of interest.

Chapter 6

Deductive Causal Theories

Use of the STRIPS assumption has made operators unacceptably difficult to describe in previous classical planners (see the example below). One of the primary reasons for this is that all effects of an action must be explicitly stated. While the desirability of deducing context-dependent effects is obvious, previous STRIPS-assumption planners have not incorporated such a capability. There are many problems in implementing it [34], especially if an expressive formalism is used to specify the rules (simply allowing variables causes problems). Two of the most obvious problems are deciding when to apply deduction, and how to control its combinatorics once it is applied. Pednault [22] and Dean, realizing the critical importance of this problem, have recently addressed it, though not in the context of classical planners. In this chapter, we present SIPE's solution to deducing context-dependent effects. It provides a powerful capability because the domain rules are fairly expressive, permitting the constraints and quantifiers previously described as well as access to different world states, all of which would be problematic in previous STRIPS-assumption planners.

Deductive causal theories are one of the most important mechanisms used by SIPE to alleviate problems in operator representation caused by the STRIPS assumption. Separation of knowledge about causality from knowledge about actions relieves operators of much of their representational burden since deductive rules allow effects of an action to be deduced without being mentioned in add or delete lists. They permit effective representation of a *causal theory* of the domain, similar to that advocated by Dean [5], and are therefore referred to as *domain rules*. By allowing knowledge of cause-and-effect relations to be specified independently of the operators, both the operators and the planning process are simplified. Since conditional

effects are deduced, operators are applicable over a much wider range of situations. This makes it much easier to express domain knowledge as SIPE operators.

Our implementation of deducing context-dependent effects allows the truth criterion to retain much of the efficiency of the STRIPS assumption, while significantly increasing expressive power. The (strict) STRIPS assumption is that no predicate will change its truth value when an event takes place unless the event description explicitly states so. This strict assumption adversely affects the specification of operators (i.e., a planner's representation of actions or events) in complex domains. For example, it is often awkward to explicitly describe all the effects of an action [28], and one is often forced to provide an operator for every possible situation in which some action might be taken. These problems can be alleviated by the efficient deduction of context-dependent effects.

Domain rules allow expression of domain constraints within a world state, as well as permitting access to different world states. Rules that allow the former are generally called *state rules*, while rules that allow the latter are generally called *causal rules*. These are discussed in more detail later. By accessing different world states, the system can react to *changes* between two states, thus permitting effects concerning what happened during an action to be deduced even though these effects might not be implied by the final world state.

This ability to reason about changes between two states is crucial. Consider the problem of sliding boxes to the left or right on a table (whose depth is the same as that of the boxes). If some action slides a box from the left edge of the table to the right edge, then any intervening boxes will have been pushed off the table. Suppose the operator describing such an action states, as its only effect, that the moved box is located at the right side of the table after the action. Rules that can only access the final state would find nothing wrong with a second box being located in the middle of the table after the action — there are no inconsistencies. A causal rule in SIPE, however, could notice that the transition between the two states would involve the block in the middle and deduce that this block must be pushed off the table.

6.1 A Motivating Example

In this section, we use a simple block-world example to introduce SIPE's domain rules and show the limitations of the strict STRIPS assumption. Consider the standard block world,

with the small extension that some big blocks are large enough to support more than one small block. A strict STRIPS-assumption Puton operator that represents moving block A from X to Y, in a world where a block can support only one block, includes explicitly listed effects of $(On\ A\ Y)$ and $(Clear\ X)$. In our extended block world, $(Clear\ X)$ may or may not be true, depending upon whether X supports blocks other than A. The solution in strict STRIPS-assumption systems is to have two operators, one representing the move of a block that is the only block on its support, and the second representing the move of a block that is one of many on top of another block. This presupposes the ability to express and test the condition of two different blocks being on another block (which may be represented by an unbound variable at the time of the test), which not all planning systems can do.

The solution in SIPE is a single Puton operator (shown in Chapter 3) that lists $(On\ A\ Y)$ as its only effect, a causal rule that deduces $\neg(On\ A\ X)$, and a state rule that deduces $(Clear\ X)$ when A is the only block on X. The system is easily capable of expressing such rules, which are shown in Figure 6-1. These domain rules relieve every operator in the system of the responsibility of deleting *On* predicates and adding *Clear* predicates, thus simplifying the description of many operators.

We shall briefly outline the use of these operators. A causal rule such as Not-on (see Figure 6-1) is applied whenever an action being inserted in a plan has an effect which matches the predicate given as the rule's trigger. The trigger is matched in the state that exists after the action is executed, while the precondition is matched in the previous state. If the precondition matches, then the effects of the rule can be added as deduced effects of the action. Thus, in the Not-On causal rule, *object*3 is bound to the support of *object*1 before it was moved to *object*2, and the deduced effect of $\neg(On\ object1\ object3)$ is added to the effects of the action. This deduced effect will then match the trigger of the Deduce-Clear state rule whose condition is matched in the current state. Because *object*4 is constrained to be in the existential class (scoping rules interpret the condition predicate as $\neg\exists object4.(On\ object4\ object6)$), the condition will match and $(Clear\ object6)$ will be deduced if there is currently no object on *object*6.

As more complex domains are represented, it becomes crucially important to use causal theories so that operators do not have to encode such knowledge. The simplifies the operators and allows them to be more widely applicable since the causal theory allow the deduction

Causal-Rule: Not-On
Arguments: object1,object2,object3;
Trigger: (On object1 object2);
Precondition: (On object1 object3);
Effects: ¬(On object1 object3);

State-Rule: Deduce-clear
Arguments: object5,object6, object4 Class Existential;
Trigger: ¬(On object5 object6);
Condition: ¬(On object4 object6);
Effects: (Clear object6);

Figure 6.1: SIPE Domain rules

of conditional effects. This can reduce exponentially the number of operators required by the strict STRIPS assumption which needs a distinct operator for every different context in which the operator may be used (where one context is *different* from another if it requires different predicates to appear in either the add or delete lists). If an action can affect N predicates, a strict STRIPS assumption might require 2^N operators to represent it, while only one operator and N domain rules are needed in SIPE. Furthermore, these domain rules will in general be used by other operators. Thus a second action that might change the same N predicates would require one operator and no new domain rules in SIPE, while it would require another 2^N operators using the strict STRIPS assumption.

6.2 Domain rules

By using SIPE's domain rules, rather than providing a full-fledged logic for deduction, we maintain strict control over the deductive process, thus helping to prevent a combinatorial explosion. At the same time, domain rules provide a fairly rich formalism for deducing effects of an action because they can include the system's entire repertoire of tools: constraints, conjunction, access to different world states, a limited form of disjunction, and limited forms of both existential and universal quantification. The control of deduction results from the above mentioned restrictions on the representation, the unification algorithm, the use of triggers, and the efficient truth criterion. The latter two depend upon the use of domain

rules to deduce context-dependent effects as described in this chapter.

First, we describe the truth criterion and its interaction with *deduced effects*. SIPE performs all deductions that it can when a new node is inserted in the plan; i.e., it computes the deductive closure of the domain rules. The deduced effects are recorded and the system can then proceed as if all the effects had been explicitly listed in the operator. Deductions are not attempted at other points in the planning process (except that they may be recomputed after new ordering constraints are added). This eliminates the necessity of deciding when to use deduction and of keeping track of which deductions have been attempted, as well as allowing the basic STRIPS algorithm to underlie the truth criterion.

SIPE domain rules have *triggers*, *preconditions*, *conditions*, and *effects*. The trigger controls rule application because it must match an effect of the node of current concern (i.e., the one for which we are deducing effects) before the rule can be applied. Deducing effects from a rule is a simple process: if the precondition and condition of a rule hold, the effects of the rule can be added as effects of the node (unless they directly contradict effects already on the node — see below). The trigger, precondition, and condition are matched exactly like any other formula during the planning process, taking advantage of the system's efficient *truth criterion*. *Conditions*, like triggers, are always matched in the current world state, while preconditions are matched in the previous state.

This is easier to express formally, so we will borrow some notation from the situation calculus. The predicate $holds(\tau, s)$ states that the formula τ is true in the state s, and the function $result(e, s)$ is the state that results after event e occurs in state s. Given a causal rule with a trigger τ, a precondition ϕ, a condition χ, and effects ψ, the following formula describes its meaning for all events and states:

$$\forall e, s \;.\; holds(\tau, result(e, s)) \wedge \neg holds(\tau, s) \wedge holds(\phi, s) \wedge holds(\chi, result(e, s)) \supset holds(\psi, result(e, s))$$

Note that domain rules do not trigger on every formula in $result(e, s)$, only on those that were not true before e occurred. This is determined efficiently in SIPE because possible triggers are simply the effects of a node.

For example, assuming an event occurs in state s1 which results in state s2, the Not-On causal rule would be expressed more formally as follows (abbreviating object as obj):

$$\forall obj1, obj2, obj3 \;.\; holds(On(obj1, obj2), s2) \wedge \neg holds(On(obj1, obj2), s1) \wedge holds(On(obj1, obj3), s1) \supset holds(\neg On(obj1, obj3), s2)$$

Domain rules can be divided into causal rules and state rules, primarily to provide another tool for the user. The causal rules are applied first because, using Georgeff's proposed view of causality [7], one might view a causal rule as representing another action that is caused by the original one and, that, for our purposes, occurs simultaneously. Considering the disappearance of a moved block from its original position as a separate event caused by the move is reasonable in a world in which some objects may leave copies of themselves behind when they move. Once the causal rules are used to determine all the simultaneously occurring events caused by the current event, the state rules then compute the domain constraints that must be true of all these events.

Note that there is no difference between causal rules and state rules other than their order of applicability - they have identical expressive power. Domain rules can be declared as state rules, causal rules, or both. In all domains yet implemented in SIPE, state rules never have a precondition (only a condition) while causal rules always have a precondition (and perhaps also a condition). Thus the causal rules are reacting to changes between states, while state rules are enforcing constraints within a state. In order to provide more power and flexibility, these limitations are not enforced as definitions of causal and state rules.

Initially, all causal rules whose trigger matches a node-specified effect are applied, thereby producing an additional set of [deduced] effects for that node. After all such rules have been applied, the system determines which newly deduced effects were not already true in the given situation and permits the causal rules to trigger on these recursively. This process continues until no effects are deduced that were not already true, thus computing the deductive closure of the causal rules. This process is then repeated for the state rules, initially firing them on all node-specified effects and all effects deduced from causal rules. In this way the deductive closure of all domain rules is computed while maintaining control of the deductive process and preventing deductive loops. (To get full deductive closure when some causal rules trigger on effects deduced by state rules, the user has two options: clump all domain rules together as either state or causal rules, or have certain domain rules be both state and causal rules.)

6.3 Problems

There are two problems in the foregoing scheme. The first problem occurs while repeatedly applying the domain rules to produce their deductive closure: a later rule might deduce a predicate that negates a predicate deduced by a previous rule. Which of these conflicting deductions should the system allow to stand as an effect? SIPE's default is to accept the first deduction and ignore the second (though it prints a message as such conflict often signifies a bug in the user's domain rules). However, there is a situation, first observed in the mobile-robot domain, in which it is desirable to deduce effects that would be conflicting if they were to be matched directly against each other but become a valid, nonconflicting representation of effects when they are recorded in the order they are deduced. This situation involves the deduction of a predicate with a universal variable that negates particular nonuniversal instances of the predicate that have already been deduced. This is exactly the not-same universal that can terminate the truth criterion (see Chapter 5), and is permitted by the system because it appears to be of general utility.

The truth criterion matches formulas against effects in the order the effects are listed. Thus, one can initially deduce $On(object1\ object2)$, where the variables are not universals, and later deduce $\neg On(object1\ object3)$, where $object3$ is universal. These deductions will be recorded in the given order, which effectively encodes the fact that $object1$ is on $object2$ only. More precisely, any formula of the form $On(object1\ X)$ will match positively if X can be constrained to be the same as $object2$, and negatively in all other cases.

The second drawback of SIPE's design is more serious. SIPE may have to instantiate variables to match the precondition of a domain rule. However, it may not be desirable to do this since the instantiation so forced may prevent a solution to the problem from being found. For example, suppose Block-Deduce-Clear is a state rule just like Deduce-Clear except that the variable $object6$ is replace by $block1$. This is an acceptable description of the domain since only blocks become clear; the table is always clear. (As written, Deduce-Clear simply never matches with $object6$ being the table since something is always on it.) But now suppose the problem is to achieve $On(redblock1\ blueblock1)$ where the two blocks are left as variables. After planning an action to move $redblock1$, application of Block-Deduce-Clear would cause the truth criterion to return constraints (when matching the condition) that would constrain $redblock1$ to be a block that was on another block. As explained below, SIPE would *not* post

such constraints in this situation. Posting them would prevent *redblock*1 from matching a block that was on the table, and later planning may discover that only a block from the table can solve the problem.

SIPE provides a few tools that help solve this problem. The user can choose whether or not to permit (in all cases) the forcing of instantiations by the application of domain rules. The default in SIPE, which has been used in all its applications, is a useful compromise. The system will constrain variables in an attempt to match a domain rule, but only when the two variables are already constrained to be of the same class. If a domain rule requires further specification of a variable's class, it will fail: it is assumed the user did not intend the deduction in this case. This is not a permanent decision. Since deductions are recomputed at each new planning level, they may change to reflect the further addition of constraints. Using this heuristic, the user can control the forcing of instantiations by the classes used in domain rules.

For example, the Block-Deduce-Clear fails in SIPE whenever it must constrain a variable representing an object to be a block because the classes are different, although it may match later in the planning after the object variable is further specified to be a block. Deduce-Clear, as shown in Figure 6-1, is appropriate for both blocks and tables, and does not force variables towards either. However, the user might want to force things to be moved off of blocks in order to clear them, as this is a good heuristic for many block-world problems. This can be done in SIPE by inserting "class blocks" after *object*6 in Deduce-Clear (or "class objects" after *block*1 in Block-Deduce-Clear). *Object*6 would then be constrained to be in both the block and object classes, and the system would constrain variables of either class in an attempt to apply Deduce-Clear. This shows the flexibility our scheme provides the user, but care is required to avoid undesirable instantiations.

6.4 Heuristic Adequacy and Expressive Power

Our claims regarding efficiency rest on the performance of the system on actual problems.[1] In all the domains implemented to date, deductive causal theories have proven useful and

[1]The representation provided by SIPE is powerful enough to write domain rules that generate large and inefficient constraints. Therefore it is not possible to prove impressive lower bounds on computation for any domain rule a user might write.

effective. Block-world problems that permit more than one block to be on top of another are solved in one or two seconds on a Symbolics 3600 using the domain rules presented in this chapter, providing a scale for our claim of an efficient truth criterion. Though corresponding data has not been published for other planning systems, we know of no other system that can solve these problems in a few seconds.

The mobile-robot domain is a much larger problem which shows both the system's heuristic adequacy and the importance of causal theories in the use of the planner. This domain contains 25 domain rules, 5 of which are causal rules, that operate over four abstraction levels on a world description consisting of hundreds of predicates. The planner produces primitive plans that provide commands, executable by Flakey, for controlling the robot's motors. This low level of abstraction requires the planner to generate hundreds of goal nodes for a plan — just to generate one plan, not to search through alternatives — yet SIPE takes about 30 seconds to completely formulate such a plan (or 9 seconds for an executable plan if the planner intermingles planning and execution). This is acceptable performance as the robot requires several seconds to move down the hall.

Causal theories are critical to achieving this level of performance. The node in Figure 6-2 is from a robot-world plan for delivering an object to an agent: in this case, a bagel to Leslie. The prominence of the causal theory in the planning process is indicated by the fact that 73% of the CPU time spent on this problem was spent on deducing effects. The only effects listed in operators for this action of going through a door are that Flakey is now at Loc11 and in Leslie's office. All of the following effects were deduced from domain rules (described in the order they appear): the rye bagel which Flakey is holding is now also at Loc11, Flakey is now next to some subset of objects, Flakey is not next to any other object, Flakey is not at any other location, Flakey now occupies an adjacent location to some subset of locations (the members of which are specified by constraints on the universal variable), Flakey does not occupy an adjacent location to any other location, the bagel is not at any other location, the bagel occupies an adjacent location to some subset of locations, the bagel is no longer adjacent to Loc11, the bagel is in Leslie's office, and neither Flakey nor the bagel is in the hallway anymore.

It is hard to make efficiency comparisons, as it appears no domain-independent AI planning systems have been tried on a problem of similar complexity, in most cases because such a

Process: P6948
Action: Go-Thru-Door;
Effects: (At Flakey Loc11),(Inroom Flakey Leslie-Office);
Purpose: (Deliver1 Leslie Rye-Bagel Flakey);
Deduce: (At Rye-Bagel Loc11),
(Nextto Flakey Object2-n6797) n6797 universal,
¬(Nextto Flakey Object2-n6802) n6802 universal,
¬(At Flakey Location2-n6782) n6782 universal,
(Adjacent-Loc Flakey Location3-n6788) n6788 universal,
¬(Adjacent-Loc Flakey Location4-n6790) n6790 universal,
¬(At Rye-Bagel Location2-n6816) n6816 universal,
(Adjacent-Loc Rye-Bagel Location3-n6822) n6822 universal,
¬(Adjacent-Loc Rye-Bagel Loc11),
(Inroom Rye-Bagel Leslie-Office),
¬(Inroom Flakey Jhall), ¬(Inroom Rye-Bagel Jhall);

Figure 6.2: Node in Robot Plan

problem cannot be effectively handled. Certainly, previous classical planners could not have encoded this domain because of the exponentially large number of operators required when there is no deductive causal theory. Unfortunately, research on more expressive non-STRIPS assumption planners rarely provides data on performance. Indeed, the planners are often never implemented, and our experience shows that implemented planners can take minutes to hours to solve even simple problems. Planners that use frame axioms or circumscription, instead of the STRIPS assumption, to solve the frame problem are faced with combinatorial problems and currently have no hope of producing a plan of this complexity in a matter of seconds. We know of no planning system that approaches the speed of SIPE on a problem as complex as this.

Causal theories are also crucial to the expressive power of SIPE. As we have seen, they permit representation of the robot domain by avoiding the exponential number of operators needed by classical planners without causal theories. By expressing knowledge of causality independently of the operators, operators become transparent, modular, and simpler. The power of domain rules is determined by the types of formulas that can appear in their conditions, preconditions, triggers, and effects. SIPE is capable of expressing more powerful formulas than other classical planners because of its constraints, quantified variables, and

ability to access different world states.

Of course, all the limitations mentioned in Chapter 2 are still present. Approaches to reasoning about action that use unrestricted logics, e.g., general frame axioms [21] and circumscription [20], provide significantly greater expressiveness than SIPE. However, they suffer from inherent computational difficulties, the need to write many axioms with all the details right, and possibly other problems such as unintended models [11] or the computation of all possible effects an action might have. With an expressive logic there is generally a need to specify axioms to deduce that all things not mentioned have stayed the same, unless the STRIPS assumption or something similar is employed. The user must, of course, implement an adequate causal theory in SIPE, but this should in general be easier to do that writing all the axioms required by one of the above systems. While the above formal systems lend themselves to rigorous analysis and hold promise, SIPE's approach has many computational advantages over them. To those who want to solve actual planning problems, the existence of an efficient implementation is important.

Chapter 7

Hierarchical Planning as Differing Abstraction Levels

It is generally recognized that planning in realistic domains requires planning at different levels of abstraction [14]. This allows the planner to manipulate a simpler, but computationally tractable, theory of its world. The combinatorics of concatenating the most detailed possible descriptions of actions would be overwhelming without the use of more abstract concepts. This has resulted in numerous hierarchical planning systems. However, hierarchical levels and hierarchical planning mean quite different things in different planning systems.

In our view, the essence of hierarchical planning (and a necessary defining condition) is the use of different levels of abstraction both in the planning process and in the description of the domain. An *abstraction level* is distinguished by the granularity [14], or fineness of detail, of the discriminations it makes in the world.[1] From a somewhat more formal standpoint, a more abstract description (in whatever formalism is being used) will have a larger set of possible world states that satisfy it. When less abstract descriptions are added, the size of this satisfying set diminishes as things in the world are discriminated in increasingly finer detail. In complex worlds, these abstract descriptions can often be idealizations. This means that a plan realizable at an abstract level may not be realizable in a finer grain (i.e., the satisfying set might reduce to the null set). For example, one might ignore friction in an

[1] Some authors use abstraction to refer to any lack of detail in a plan, even if the granularity remains unchanged, e.g., the omission of some ordering constraints from a plan.

abstraction of the domain, but find that the abstract plan cannot be achieved at the lower abstraction level when the effects of friction are included in the world description.

To see how hierarchical planning can help avoid the combinatorial explosion involved in reasoning about primitive actions, consider planning to build a house. At the highest abstraction level might be such steps as site preparation and foundation laying. The planner can plan sequences of these steps without considering the detailed actions of hammering a nail or opening a bag of cement. Each of these steps can be expanded into more detailed actions, finally getting down to the level of nail-driving, but with the abstract plan eliminating all but a few of the possible nail-drivings at each point where the plan could drive a nail. Hierarchical abstraction levels provide the structure necessary for generating complex plans at the primitive level.

7.1 The Many Guises of Hierarchical Planning

The planning literature has used the term "hierarchical planning" not only to describe levels of abstraction, but also to describe systems containing various hierarchical structures or search spaces, metalevels, and what we will call *planning levels*. Examples of each of these are given below. Planning levels are of particular importance because confusing them with abstraction levels causes a problem in various implementations of hierarchical planning, particularly in classical AI planners (see below).

Many planners produce hierarchical structures (e.g., subgoal structures) during the planning process or explore hierarchically structured search spaces. Generally having nothing to do with abstraction levels, they occur even in nonhierarchical (by our definition) planners that allow only one level of abstraction. STRIPS, while nonhierarchical, could be regarded as producing plans with a hierarchical structure, e.g., its triangle tables. The Hayes-Roths [13] use the term *hierarchical* to refer to a top-down search of the space of possible plans where more abstract plans are at the top of this search space. This involves a hierarchical search space that contains abstraction levels, but the levels in the hierarchy are not defined by these abstraction levels.

Hierarchical planning is also used to refer to metaplanning. Reasoning at a metalevel involves reasoning about the planning process itself. This is an entirely different domain,

not merely an abstraction or idealization of the original domain. Stefik [29] states that ". . .layers of control (termed *planning spaces*) . . . are used to model hierarchical planning in MOLGEN". In this case, the three planning spaces are being used to implement metaplanning and, respectively, represent knowledge about strategy, plans and genetics; the first two are not abstractions of the genetics domain. (MOLGEN does provide for planning at different levels of abstraction through its constraints.)

The above uses of the term "hierarchical planning" describe processes or structures unrelated to the use of abstraction levels. Therefore, any confusion generated is terminological and is not an indication of possible conceptual problems within the planning system itself. Planning levels, on the other hand, have been confused with abstraction levels – which, as we shall see, can lead to problems within the planner, particularly if the planner incorporates the STRIPS assumption [34]. *Planning levels* are artifacts of particular planning systems and may vary considerably from planner to planner. They are not defined by a different level of abstraction in the descriptions being manipulated, but rather by some process in the planning system. Most planning systems have some central iterative loop that performs some computation on the plan during each iteration. This may involve applying schemas, axioms, or operators to each element of the existing plan to produce a more detailed plan. To the extent that such an iteration takes one well-defined plan and produces another well-defined plan, we will call it a planning level. Planning levels may correspond, in some systems, exactly to the hierarchical structures discussed earlier, but this is coincidence. They are defined by the planning process, not data structures, and may or may not correspond to hierarchical data structures within a particular system. The term is admittedly vague but in classical AI planning systems, and many others as well, it has a precise definition.

All classical AI planners have distinct and well-defined planning levels. In these systems, a new planning level is created by expanding each node in the plan with one of the operators that describe actions. In the literature these levels are often referred to as hierarchical, which implicitly associates them with abstraction levels. In fact, they are independent of abstraction level; a new planning level may or may not result in a new abstraction level, depending upon which operators are applied. For example, in the blocks world described by Sacerdoti [27], there is only one abstraction level. Cleartop and On are the only predicates and each new planning level simply further specifies a plan involving these predicates. Thus, all the hierarchical levels in the NOAH blocks world are actually planning levels, that result

in adding further detail to the plan at the same abstraction level. Others have described such an omission of detail as an "abstraction", but we specifically require an abstraction level to involve different predicates with different grain-sizes.

Planning systems not in the classic tradition also have planning levels. Rosenschein's planning algorithm, using dynamic logic [24], attempts to satisfy a set of planning constraints. Running his "bigression algorithm" on each constraint in the set (which he claims is a straightforward extension of his system) would constitute a planning level. Agenda-based planning systems also have natural planning levels that are defined by the execution of one agenda item. These planning levels would be somewhat different from the others we have discussed, as they might not involve performing some operation on each element of a complete plan. Consequently, they are not likely to be confused with abstraction levels.

7.2 A Problem with Current Planners

There is a problem that can arise when planning and abstraction levels are interleaved in a planner making the STRIPS assumption, as they are in classical planning systems. This problem exists in many NOAH-tradition planners but has never been documented.

In such a planner, one element of the plan can attain a lower abstraction level than another element at the same planning level, depending upon which operators are applied. Thus, the plan at the current planning level could be **P1;Q1;G**, where **P1** is an action making the predicate P true, **Q1** an action making the more abstract predicate Q true, and **G** a goal that depends upon the truth of P for its achievement. (We use the symbol ; to denote sequencing of actions.) With the STRIPS assumption, the planning system will find that P is true at **G**, since **Q1** does not mention changes involving any less abstract predicates (such as P). In fact, the truth of P may depend upon how **Q1** is expanded to the lower abstraction level, since it may or may not negate the truth of P. Thus, conditions may be evaluated improperly in these planners, resulting in incorrect operator applications.

For these planners to correctly apply their truth criterion at time N in the plan, they must ensure either that all relevant information at the proper abstraction level is available for the actions preceding N, or that subsequent expansions to lower abstraction levels will not change the truth value of the query. However, many existing planners (e.g., NOAH and SIPE) do

not provide this assurance. We might say such planners exhibit *hierarchical promiscuity*, but there are good reasons for being promiscuous. Various solutions to the above problem are discussed below, but the most straightforward one is to impose a depth-first, left-to-right planning order that is sensitive to change in abstraction level. This is the approach taken by ABSTRIPS [26]. This means that, when a condition with predicates of abstraction level M is checked at time N in the plan, all possibly relevant information at abstraction level M or higher will be available for every plan element occurring prior to N.

However, this is not always desirable since many advantages can be gained by planning certain parts of the plan expedientially (or opportunistically) to lower abstraction levels. For example, in planning a trip from Palo Alto to New York City, it might be best to plan the details of the stay in New York first, as this could determine which airport would be best to fly into, which in turn could determine which Bay Area airport would be the best departure point. Thus, we do not want to restrict ourselves to depth-first, left-to-right planning, which would require choosing the Bay Area airport before the one in New York.

While the foregoing problem will be discussed in this paper in our terminology of "operators" and "goals", it applies equally well to any planner that must coordinate deductions over different abstraction levels. For example, Rosenschein's hierarchical planner based on dynamic logic [24] must address this issue in order to produce correct plans. The definition of this problem will be made more concrete by looking at it in the indoor mobile-robot domain.

7.2.1 Abstraction Levels in the Robot Domain

It was the encoding of the mobile-robot domain in SIPE that revealed the problem of coordinating abstraction levels. In our encoding of this domain, the most abstract level of the planning process reasons about the tasks that can be performed, such as preparing a report or delivering an object. Our simple domain requires only one level for this, but more complex tasks might require several abstraction levels to describe them. The first level below the task level (referred to as the Inroom level since *Inroom* is the crucial predicate) is the planning of navigation from room to room. This plans a route that may require many planning levels for all the necessary operators to be applied, but it does not involve any reasoning about particular doors or locations. High-level predicates describing connections indicate that it is reasonable to move from one room to another, but without first considering any details as to

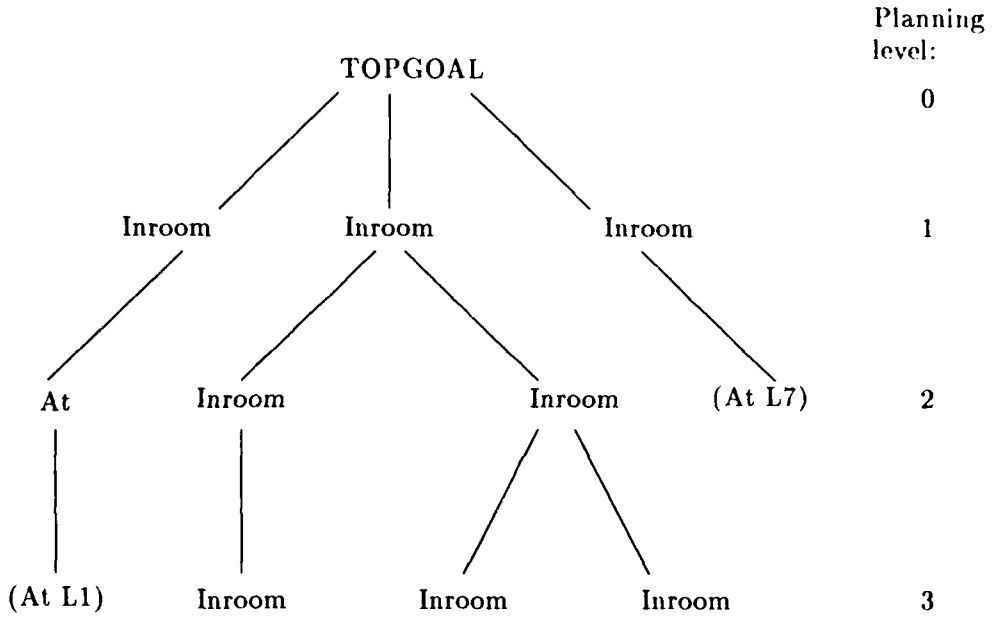


Figure 7.1: Hierarchical Plan in Robot Domain

how this might be done or whether it might even be possible in the current situation. When such a move is planned to a lower abstraction level, it may fail or many actions may have to be performed to clear a path.

Below the room level is the Nextto level, which plans movements from one important object (that the robot is next to) to another. For example, to copy a paper, the robot will have to get next to the door of the copy center, then pass through the doorway, then get next to the desk of the operator, etc. This abstraction level plans high-level movements within a room but is still not concerned with actual locations — *Nextto* is the crucial predicate.

The lowest level is the location level, where SIPE plans movements down to the level of the actual locational grid it has been given. This may involve planning to move obstacles so as to clear particular paths. *At* is the crucial predicate at this abstraction level.

7.2.2 Coordinating Abstraction Levels

The problem of coordinating abstraction and planning levels arose in the plan depicted in Figure 7-1. The initial goal produced three Inroom goals at the first planning level. Of these three subtrees, the first and last were transformed into the lowest abstraction level at planning level 2 (intervening Nextto goals have been disregarded for the sake of simplicity). The middle subtree, however, is still at the room abstraction level on planning level 3, as it required several operator applications to find a path through the rooms to accomplish its goal. This hierarchical promiscuity leads to trouble.

Now, when the planner applies an operator to the (At L7) goal at planning level 2, any precondition that queries the At predicate will find (At L1) to be valid, since that is the last place at which At is affected and the STRIPS assumption assumes all actions leave predicates unchanged unless they explicitly specify otherwise. But this is not correct, as At may be affected after the middle subtree is expanded to the lowest abstraction level. If the operator depends critically on the value of the At predicate, its application to (At L7) may be incorrect. The resulting plan will have commands that move the robot from location L1 to location L7, whereas the robot is not likely to be at L1 when this part of the plan is executed (because of movements made in the middle subtree). Whether such an operator application will prevent the correct plan from being found depends on how a particular planner detects invalid plans and how the search space is organized.

7.3 Solutions

There are many alternative solutions to this problem. They range from calculating all possibly relevant information before it may be needed to ignoring the problem altogether and simply letting the user beware of the consequences. The former, and most straightforward, is to force the planner to plan in temporal order and to provide for calculation of all the predicates at one abstraction level that may be needed later. Many planners do this, though it is usually not made clear that they depend on several assumptions to avoid subtle problems such as this one. ABSTRIPS [26] is an example of planners that use this approach. It assigns abstraction-level numbers to the predicates and plans a lower level only when all necessary computations have been made. SIPE provides, as a user-selectable option, the ability to

control hierarchical planning in this manner. This is useful for performance comparisons with the different techniques described below.

The problem with this approach is that the planner is limited in the order in which it can process goals. Quite often the order imposed will not be optimal (as in the Palo Alto-to-New York example). The flexibility to plan certain parts of the plan expedientially to lower abstraction levels is lost. Constraints generated during such lower-level planning can narrow down the search, thus resulting in potentially large gains in efficiency. For these reasons, planners like SIPE, NOAH, and NONLIN allow the mixing of planning and abstraction levels.

The approach used in these latter planners is susceptible to the level coordination problem and therefore requires the user to be alert. Incorrect checking of conditions similar to the one described can occur in these systems, depending on the task and the encoding of the operators. This can result in incorrect operator applications. The planning system may have mechanisms that later detect the plan is incorrect, as SIPE and perhaps NONLIN do. However, the proper solution may not be found if the operator applied incorrectly is the one actually needed for the correct solution, since it may not be retried. Its premature application should have been delayed one or more planning levels so that other parts of the plan could be planned to a lower abstraction level. The user is responsible for writing operators that will accomplish this delay. This process is facilitated by certain features of SIPE, as we shall see.

It is appealing to look for a technique between the inefficiency of computing everything in a certain order, and the expediential behavior of systems that perhaps miss valid solutions. This would involve reasoning about what properties will remain invariant during further planning of certain goals. For example, in Figure 1 the planner might calculate whether the possible expansions of the middle subtree will affect the value of the At predicate. If not, planning can proceed expedientially. This is all the more appealing because reasoning about concurrency depends on determining invariant properties of a sequence of actions.

Despite the attraction of this approach, there are severe difficulties entailed in computing these invariances. It would be best if they could be computed automatically from the operators without requiring the user to supply additional knowledge. (The STRIPS assumption effectively makes this computation for predicates at one abstraction level, but the

computation must be done for predicates at other abstraction levels.) In general, this is not computationally tractable. It is similar to the problem of regressing conditions through actions ([24],[34]), except that the regression must be done through every possible expansion of the actions for an indeterminate number of planning levels. Furthermore, simple schemes that simply check for predicate names that may possibly be changed will probably find very few invariances. For example, in the robot domain, every high-level goal will alter the values of At predicates at the lowest level. More sophisticated schemes that check possible values for the arguments of the predicates, perhaps determining ranges of values they might acquire in all possible expansions, would themselves require solving a large search problem.

An alternative is to have the user provide information about what remains invariant over actions. While this may be useful for some domains, in general the same criticisms made above apply to this case. There will in general not be many things (at lower abstraction levels) that are invariant; moreover, it may even be difficult to indicate explicitly what they are, as the invariance may involve complex constraints on the allowable arguments to predicates. In addition, one of the chief advantages of abstraction levels is that specifying details is unnecessary at higher levels. Computing invariants would require information as to which lower levels are affected in what way by each higher-level goal, thus removing some of the advantage gained by not planning at the lowest level from the very beginning. The computational costs of this can quickly become overwhelming as we shall see.

7.3.1 Delaying Operator Applications in SIPE

For reasons given above, SIPE is hierarchically promiscuous and gives the user the burden of encoding the domain in such a way that incorrect evaluation of conditions will not produce applications of operators that prevent solutions from being found. We have solved this problem within SIPE for the robot domain in two different ways (in addition to the option of using the ABSTRIPS solution). One solution involves delaying the application of certain operators and the other solution involves the introduction of certain less abstract predicates at earlier planning levels.

The first solution involves a novel use of operators, developed during implementation of the robot domain, that effectively delays the achievement of certain goals until the appropriate juncture, as long as the latter can be ascertained by conditions that are expressible

Operator: Not-yet
Arguments: robot1,location2,area1,location1;
Purpose: (At robot1 location2);
Precondition: (At robot1 location1),
(Inroom robot1 area1),
¬(Contains location1 area1);
Plot: COPY
End Plot End Operator

Figure 7.2: Operator for Delaying Operator Application

as preconditions of a SIPE operator. This is best shown by returning to the robot domain example.

In the SIPE robot domain, only the planning of At goals is affected when abstraction levels vary in the plan. Figure 7-1 depicts the type of situation in which the accomplishment of an At goal must be delayed. This is done by using the operator shown in Figure 7-2. It delays the solving of At goals until the part of the plan preceding them has been brought to the same level of abstraction. This is done by checking whether the At location of the robot is in the same room as its Inroom location. If the precondition of this operator matches, it means that the last At predicate specified as an effect of an action came before the last Inroom predicate specified as an effect. Consequently, the latter action must still be planned to the lower level of abstraction. [2]

This operator is applied before any other to an At goal. The plot of Not-yet is simply the token **Copy** that copies the goal from the preceding planning level. It is necessary to use a special token rather than specify the At goal in normal syntax. Normally SIPE inserts the precondition of an operator into the plan and maintains its truth. In this case the precondition will not be true in the final plan, so the copy option inserts the appropriate goal without first inserting the precondition. With this feature and the above operator, SIPE can mix abstraction and planning levels freely in the robot domain without missing a solution on our test problems. However, there may be problems in the domain that cannot be solved without creating additional delaying operators.

[2] Of course, this operator could still be fooled if you planned a circular route that ended in an Inroom goal for the same room that contained the last preceding At location. However, this would cause a problem only if the eventual location reached in the expansion of the Inroom goal were different from the one in the earlier At goal. This situation never arises in our domain.

7.3.2 Introducing Low-Level Predicates

Instead of using the Not-yet operator, the second solution involves introducing lower-level predicates at higher abstraction levels so that the truth criterion will still be correct. In this case, we add a lower-level At predicate (which includes an uninstantiated locational variable) to every higher-level Nextto goal as a placeholder for the predicate that would be produced during some future expansion. When the Nextto goal is expanded to the lower level, the location actually reached will eventually become the instantiation of the locational variable introduced. At the planning level of the Nextto goal, any At predicate in a condition being tested will match with the newly inserted At predicate, preventing the planner's incorrect assumption that the Nextto goal does not affect the truth-value of the condition.

This solution takes advantage of SIPE's ability to post constraints on variables. The newly introduced At predicates effectively document the fact that the At location may eventually change during any expansion of the Nextto goal (even though the location is not yet known). Before and during such an expansion, the location variables can accumulate constraints on their possible values, so the planning process will not be hindered. The matching of conditions will always be correct because these At predicates are present everywhere the At location might change.

Incorporating this change into the SIPE operators written for the first solution was easy. Only three of the 25 operators posted Nextto goals, so only those three had to be changed. The process of converting these three operators is illustrated by contrasting the original Fetch operator, described in Chapter 3 and shown in Figure 7-3, with the Fetch operator including At predicates, shown in Figure 7-4. In the plot, each goal node and process node with a Nextto predicate in its effects is given an additional effect that is an At predicate involving a new locational variable. The latter is included in the arguments of the goal or process so as to permit appropriate matching with the variables in the operators that solve Nextto goals. The locational variable is added to the arguments of the operator, whose precondition can also specify any predicate that constrains the variable. In particular, the variable should be constrained by its containing room.

Once the three operators were converted in this manner, SIPE was able to solve all the problems in our test domain. This solution appears robust and should not prevent the planner from successfully dealing with any problems it might solve by means of an ABSTRIPS-like

Operator: Fetch
Arguments: robot1,object1,room1;
Purpose: (Holding robot1 object1);
Precondition: (Inroom object1 room1);
Plot:
    Goal: (Inroom robot1 room1);
        Protect-until: (Holding robot1 object1);
    Goal: (Nextto robot1 object1);
    Process
        Action: Pickup;
        Arguments: robot1, object1;
        Effects: (Holding robot1 object1);
End Plot End Operator

Figure 7.3: Original Fetch Operator

Operator: Fetch
Arguments: robot1,object1,area1,location1;
Purpose: (Holding robot1 object1);
Precondition: (Inroom object1 area1),
    (Contains location1 area1);
Plot:
    Goal: (inroom robot1 area1);
        Protect-until: (Holding robot1 object1);
    Goal: (Nextto robot1 object1);
        Arguments: robot1, object1, area1, location1;
        Effects: (At robot1 location1);
    Process
        Action: Pickup;
        Arguments: robot1, object1;
        Effects: (holding robot1 object1);
End Plot End Operator

Figure 7.4: Fetch Operator with At Predicate

Problem:	Not-yet	introduce At	ABSTRIPS
Original	29.5 (7)	54.2 (7)	32.1 (11)
Shorter path	24.8 (7)	46.7 (7)	25.1 (8)
longer path	43.7 (7)	57.4 (7)	35.7 (10)

Figure 7.5: Symbolics 3600 CPU time and planning levels for solutions

approach. Characteristics of the domain are again exploited in this solution. By having to plan about less abstract entities at a more abstract level, we are giving up some of the advantage gained by planning hierarchically. However, it is reasonable in this case because we need introduce only one lower level predicate early (albeit the most important one), and we need introduce it only a single abstraction level early. There is no difficulty in coordinating any other pair of abstraction levels in this domain. However, the introduction of more variables and constraints significantly increases the effort required. Using this technique to solve problems in the robot domain takes from one-third again to twice as long as using the Not-yet operator (see next section).

7.3.3 Comparison of Solutions

The two techniques described above were tested on three different problems in the robot domain. The ABSTRIPS control regime was also used to solve these problems. (This involves using the same operators as were used while delaying operator application except that the Not-yet operator is eliminated.) The original robot problem involves a choice of different paths and entails seven planning levels for producing a primitive plan with 58 process/phantom nodes. The other two problems are similar, but one requires a shorter path to be found and the other requires a longer path. Figure 7-5 depicts the cpu time and number of planning levels required for each problem. It is to be expected that the ABSTRIPS control regime is as efficient as delaying operator application since these particular problems do not admit to simpler solution by planning later parts of the plan to a lower abstraction level. For problems and domains with this property, ABSTRIPS-like coordination of levels is preferred since it is both efficient and correct. (It does, of course, require more planning levels.)

The Not-yet solution accomplishes the delayed application of operators when necessary,

but permits expediential planning in other cases. This retains flexibility while remaining efficient by not regressing conditions through possible expansions of actions. As no lower-level predicates are introduced early, full advantage is taken of hierarchical planning. The disadvantages of this approach are that the user (though relieved of the necessity of specifying invariance properties for higher-level goals) must write appropriate delaying operators and, furthermore, must have anticipated all possible situations in which operators would need to be delayed. In complex worlds this means that novel problems might not be solved. In addition, it may not always be possible to express the appropriate delaying conditions as a SIPE precondition. In an application in which efficiency is of paramount importance and failure to solve a particular problem can be tolerated, this may be a desirable approach.

Introducing At predicates is more robust and less likely to fail on novel problems. It was surprisingly easy to implement in SIPE. However, when low-level predicates are introduced at a higher abstraction level, it is significantly less efficient. The advantages of hierarchical planning can be readily seen, as the introduction of only one predicate (albeit the crucial one) at the next higher abstraction level nearly doubles the cost of computation.

Chapter 8

Search

Development of SIPE has not addressed the issue of intelligent control of the search process, in part because searching algorithms and heuristics will need to be domain-dependent. For example, consider the advice "use existing objects", a fairly domain-independent concept that is used by Sacerdoti in NOAH [27] and mentioned by Wilensky [35] as a metagoal for metaplanning. However, this idea still involves domain knowledge. In the house-building domain, it is desirable to use the same piece of lumber both to support the roof and the sheetrock on the walls. But in another domain, this may not be a good strategy. On the space shuttle, one might want different functions to be performed by different objects so the plan will be more robust and less vulnerable to the failure of any one object. So the "use existing objects" idea makes assumptions about the domain that need to be stated. In general, because all search strategies rely on certain properties of a search space in order to function well, no one strategy can be selected for a domain-independent system.

For this reason, SIPE provides only a simple automatic search strategy, but has built the system on basic mechanisms (primarily contexts) that facilitate the ability of users to encode their own search strategies. The system's automatic search is a straightforward depth-first search with chronological backtracking, that provides for the interleaving of planning and execution. However, the representation is powerful enough to encode search-control knowledge within operators (see Chapter 3). Unlike its predecessors, SIPE is designed to also allow interaction with users throughout the planning and plan execution processes. The user is able to watch and, when he wishes, guide and/or control the planning process. This is useful for debugging, and allows users to address larger problems that may initially be beyond

the capabilities of fully automatic planning techniques. Development of an interactive planner also encourages us to deal with the issue of representing the planning problem in terms that can be easily communicated to a user. While work on SIPE has raised issues in human-machine interaction, we will discuss only the planning aspects of this effort.

8.1 Automatic Search

Although SIPE manipulates its representations efficiently, its straightforward depth-first search with chronological backtracking will obviously not perform well on large, complex problems. The poor performance of automatic search is not debilitating for two reasons: the system has been designed and built to support interactive planning, and the operators are powerful enough to express metaknowledge that can be used to effectively control the search by correctly narrowing the set of applicable operators.

The only backtracking points are alternative operators that could have been applied. In other systems, backtracking points are also generated by alternative ordering constraints and alternative variable instantiations. SIPE uses its ability to post constraints on variables to avoid instantiating a variable unless the instantiation is forced. Constraints allow the system to accumulate knowledge about the instantiation without committing to it. Nevertheless, instantiation choices still appear in two places. During application of domain rules to deduce effects of actions, it may be possible to apply a rule by making an instantiation. Chapter 6 describes the heuristic (based on class constraints) that SIPE uses in this case.

It may also be possible to accomplish a goal by making an instantiation instead of applying an operator. By default, SIPE remains true to its least-commitment philosophy by refusing to instantiate in this situation, but Chapter 9 describes options the user can select that will allow the goal phantomization critic to make instantiations when they are not forced. We are satisfied with the heuristic solution for avoiding choice points in deduction — the combinatorics of that problem demand some such heuristic. However, it may be desirable to allow choice points for the solving of goals by instantiation in domains where the problem can still be kept tractable.

The addition of ordering constraints is also not backtracked over, except that at the time the constraints are added, SIPE applies problem recognizers from the replanner (see Chapter

11) and immediately rejects the proposed ordering if serious problems are introduced (thus permitting alternative orderings to be tried). Again, it may be desirable to allow choice points for alternative orderings in domains where the problem can still be kept tractable. SIPE generally adds ordering constraints only when they are forced, but Chapter 9 describes options the user can select that will allow the goal phantomization critic to add ordering constraints when they are not forced.

The search is responsible for balancing time spent checking critics and the global constraint network with time spent planning. This balance is an open and important question, and the desired result is domain dependent. In a domain where global constraints are frequently violated, it may be best to check them after every operator application. If, on the other hand, global constraints are almost never violated, it may be best to check them only after a complete plan at the primitive level is produced. While either of these extremes can be achieved through interactive control of the system, SIPE's search implements a compromise. The global constraint satisfaction routine is called once per planning level (i.e., after one level of expansion is done to every node in the plan). This can be easily changed in domains where better performance might be achieved by investing this effort more or less often.

The search directs several operations at each planning level, and the order of these operations can be important to overall system performance. The search begins by trying to phantomize any goal in the original problem. At each planning level, it applies an operator to each open node (i.e., each nonprimitive goal, process, or choiceprocess node), copying down any other nodes (e.g., precondition nodes, primitive nodes), to produce a plan at the next planning level. The deduced effects of the nodes copied down are recalculated, since they may change in the different context specified by the more detailed plan. The search checks for problems this may cause, and may change phantom nodes back to goals, as well as rejecting operator applications that falsify preconditions already present in the remainder of the plan. The latter is an important pruning of the search space — it means that some operator already applied should not have been, so that a later search of alternatives will eventually produce a correct plan that includes the operator application currently being rejected (if such a plan exists). The critics (see Chapter 9) are then called on the new detailed plan. They make sure the global constraint network can be satisfied, then try to phantomize goal nodes, then check for resource conflicts, and finally check for problematic parallel interactions. Both of the last two operations may add ordering constraints to fix conflicts or problems. The last

operation may also insert appropriate goal nodes if parallel postconditions on join nodes are violated.

The ordering of the above operations can be important. Phantomization should be attempted before the operations that follow it, since the problem may be simplified. In general, one wants to solve the global constraint network first, since this often forces instantiations of variables which may simplify everything. However, phantomizing goals may also instantiate variables, after which one might want to check the global constraints again in case they force further instantiation. One could imagine applying these two operations repeatedly until one of them did not force an instantiation. In applications of SIPE, this situation has not come up, so the search merely calls each of these operations once, except in the case where a final [primitive] plan is produced — in which case the constraints must be rechecked to assure that the plan is still valid after phantomizations.

Chapter 9 describes these plan critics in detail and presents an example of the automatic search solving a problem, showing the ordering and frequency of these operations.

8.2 Intermingling Planning and Execution

The search also has the ability to interleave planning and execution. This will be described in detail since other classical planners do not have this ability, as critics frequently point out. Classical planners have historically planned every step of each plan to the lowest level of detail — this is the reason plans in the mobile robot domain for retrieving objects take about 30 seconds to generate. Such detailed planning can often be undesirable [8], since it prevents the planner from reacting quickly to events. Furthermore, as actions are planned further into the future, it becomes less likely that they will be useful. The probability increases that some unexpected event will render the remainder of the plan unsatisfactory.

Fortunately, there is no inherent reason that classical planners have to plan everything to the lowest level of detail, so SIPE permits intermingling planning and execution. The operator description language allows users to encode domain-specific information about which goals and actions can have their planning delayed. The user can simply include the word "DELAY" in the description of a node in the plot of an operator. The search will then not plan any such goal or action until a plan suitable for execution has been generated. The planning of

the delayed goals is started as a background job as soon as the original plan is ready for execution.

The original plan is used by the execution monitor until either an unexpected event happens or the goals whose planning has been delayed are reached. In both cases, the plan produced by the delayed planning process is retrieved (possibly waiting for the process to finish) and updated with information about nodes that have already been executed. Execution proceeds on this updated plan while another background job continues to plan any delayed goals in this new plan. When SIPE attempts to retrieve the results of the delayed planning process, it may notice that the delayed planning fails, in which case the system tries again to solve the original problem in whatever state the world is currently in.

The encoding of domain-dependent knowledge for this purpose is effective because such knowledge is generally available. For example, in the robot domain, the robot can obviously begin executing its plan to get to the object to be picked up before planning how to deliver the object after picking it up (assuming the robot does not make hallways impassable as it travels down them). Thus, the operator for fetching and delivering an object should have a delay put on its second goal. Goals should not be marked for delayed planning unless there is a high probability that they can be achieved, or it is known that their solution is independent of the solutions chosen for prior goals. The planner can begin execution with some assurance that its initial plan should be the beginning of a valid solution for the whole problem. Domain-independent criteria for delaying planning, e.g., delaying planning after a certain number of actions have been planned, would be arbitrary and would not be able to provide this assurance.

The delay described above (on the deliver goal following the fetch) is the only one introduced into the operators used to solve problems in the robot domain. With this minor addition, SIPE produces a plan for the same problem that is ready for execution in only 9 seconds (rather than 35). The remainder of the plan is usually ready in complete detail before the robot travels very far down its first hallway. This enables SIPE to react much more quickly to situations, and reduces the time spent waiting on the planner. It is also easy to envision more options than simply planning delayed goals in a background job. On the basis of domain-dependent knowledge, these goals could alternatively be planned immediately or left unplanned until execution reaches that point in the plan.

8.3 Interactive Control

The user can control the search interactively, taking advantage of SRI's Graphical Interface to view the partial plans produced as graphs. This ability is quite useful, since the system can be guided through problems that would not be solved in a reasonable amount of time with the automatic search. It is also very useful for debugging purposes.

Control is accomplished through self-explanatory menus that allow the user to invoke planning operations at any level without being required to make tedious choices that could be performed automatically. The user can direct low-level and specific planning operations (e.g., "instantiate *plane*1 to N2636G", "expand Node32 with the Fetch operator"), high-level operations that combine these lower-level ones (e.g., "expand the whole plan one more level and correct any problems"), or operations at any level between the two (e.g., "assign resources", "expand Node32 with any operator", "find and correct harmful interactions"). If the user chooses to control the planning at the lowest level, then he must call the plan critics appropriately to assure than the plans being produced are valid. Through use of the context mechanism, the user can instantly change his attention to different alternative plans.

8.4 Domain-Dependent Search Control

Since we view domain-dependent search control as necessary in complex domains, SIPE provides for its realization in several ways. This chapter has described several ways that properties of the domain can be used to control the automatic search and the interleaving of planning and execution. Chapter 3 described ways in which operators can express meta-knowledge that can be used to control the search. There are other ways in which the user can take advantage of the system's representational power to encode search control knowledge. The whole purpose of abstraction levels is to control the search, and powerful abstract operators can be written. In addition, nodes can be given extra arguments that are variables with constraints. These variables serve no purpose other than to unify with arguments of operators, which means operators used to expand such a node must satisfy the constraints posted on this extra argument. In this way, knowledge encoded in constraints can be used to control which operators are applied.

However, the primary feature of SIPE which allows sophisticated search control strategies

to be implemented is the ability to explore alternative plans in parallel. Other domain-independent planners have not provided this capability, which facilitates implementation of various search strategies, including best-first. This is implemented by the contexts and choice points described in Chapter 3. Constraints on variables are posted relative to choice points. The context is used to select those constraints on a variable that are part of the current plan. This permits the user to shift focus easily among alternatives, which cannot be done in systems that use a backtracking algorithm, in which descriptions built up during expansion of one alternative are removed during the backtracking process before another alternative is investigated. Most other planning systems either do not allow alternatives (e.g., NOAH) or use a backtracking algorithm (e.g., Stefik's MOLGEN, NONLIN). An exception is the system described by Hayes-Roth et al. [13], in which a blackboard model is used to allow the shifting of focus among alternatives.

In complex domains, it may be reasonable to build a metaplanning module to control SIPE in a domain-dependent manner. The types of search control one might implement using SIPE's context mechanism are unlimited. For example, one can imagine using Stefik's layered hierarchical approach to metaplanning with SIPE, p.ovided the lowest layer upon which the design and strategy layers will operate.

Chapter 9

Plan Critics

There is good reason for a planner to temporarily produce plans that are not valid. Allowing nonlinear plans (which is necessary for sufficient expressive power) means that plans will often not be valid until the parallel interactions are analyzed and corrected. It is often not tractable to check the global constraint satisfaction problem every time a new constraint is posted. For these reasons, it would not be computationally intractable for a classical planner to be implemented in such a way that one could prove that every single operation performed by the planner produces a correct plan. Thus, classical planners employ plan critics which periodically check the validity of the plans that are produced, and possibly modify them in response to any problems that are found.

In SIPE, there are several reasons for applying critics in addition to the obvious problem of checking helpful and harmful interactions among possibly parallel actions. As we have seen in previous chapters, constraints that are globally unsatisfiable may have been posted. Resource conflicts may need to be corrected. Some goal nodes may have already been achieved by planning operations already taken. Plan critics are therefore called to check parallel interactions, phantomize nodes, check resource conflicts, and solve the global constraint network. These critics may resolve problems they find by applying *solvers* which modify the plan. SIPE's solvers may instantiate variables, further order the plan, add new goals to the plan, change goals (which may be either goal nodes or choiceprocess nodes) to phantoms, and even remove subplans that are no longer part of an optimal solution. Critics must either solve every problem they find, or fail, which causes the automatic search to backtrack and explore other alternatives.

In Chapter 8, we discussed the issues involved in deciding how often to employ critics. Currently SIPE applies them once per planning level in the automatic search. Through interactive control this can be easily changed. In this chapter, we describe the techniques actually used by the critics to find and solve problems. The solution techniques of instantiating variables, inserting new goals, and changing goals to phantoms are simple and do not need to be discussed further. Adding ordering constraints to the plan is the most complicated solution technique, since it can introduce changes that may affect the remainder of the plan in a drastic manner — resulting in the addition of goals or the removal of subplans. It is described below, and the last section of this chapter presents examples of the automatic search applying many of these mechanisms to solve a block-world problem. The system collects all proposed linearizations and attempts to optimally order them.

Since resource reasoning is an important contribution of SIPE and involves several novel techniques, it is described in detail in its own chapter (Chapter 10). The other critics, those for global constraint satisfaction, goal phantomization, and solution of harmful parallel interactions, are described below. The resource critics described in the next chapter use the solution technique of adding ordering constraints which is described below.

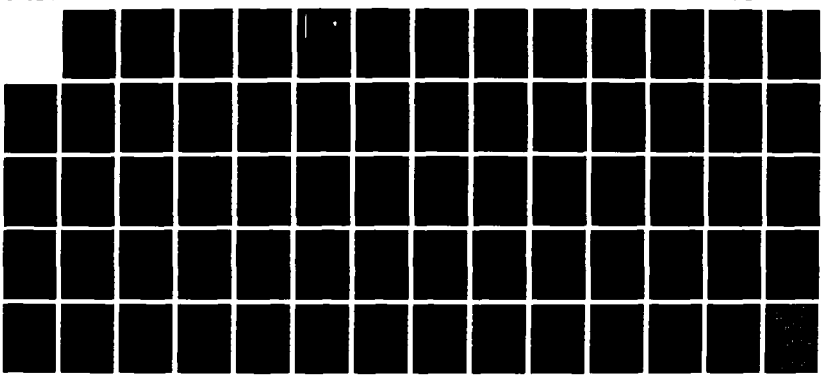
9.1 Solving the Constraint Network

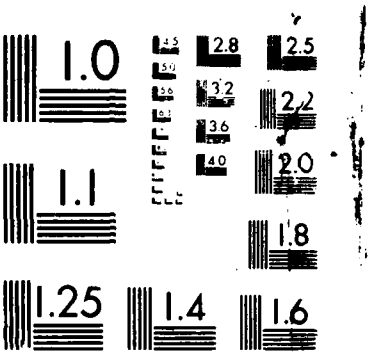
Global constraint satisfaction in SIPE is not of great interest. It uses a straightforward depth-first search to find a legal instantiation for each variable such that no constraints are violated. It actually posts instantiations only if they are forced. If there is no solution, it *simply means that the current* branch of the search must be pruned. None of the solvers are applied in an attempt to make the constraints satisfiable. Chapter 10 describes additional checks made by this critic for numerical constraints.

There are some simple heuristics for guiding the search within this critic. The order in which legal instantiations are found for variables is determined so that optional-not-same constraints will be satisfied if it is possible to do so. Constraints that are easily computed are checked first. In particular, pred and not-pred constraints are checked last. The system also saves the last solution it found for the global constraint satisfaction problem, and at each point in the search tries first the choice that worked in the previous solution. This technique eliminates most searches in the majority of cases. Since this critic is called at each planning

AD-A195 154 RESEARCH ON PROBLEM-SOLVING SYSTEMS(U) SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER D E WILKINS FEB 88 AFOSR-TR-88-0563 2/2

UNCLASSIFIED F49620-85-K-0001 F/G 12/9 NL





level, the incremental addition of constraints between calls to the critic generally results in a network that has a solution that shares many instantiations with the previous solution.

This simple algorithm has resulted in satisfactory performance in the domains currently implemented. In larger problems, particularly complex scheduling problems, it may be desirable to implement a more sophisticated algorithm. SIPE currently does not relax any of its constraints — another ability that would be useful in more complex domains.

9.2 Parallel Interactions

A least-commitment philosophy involves not committing to orderings unnecessarily. We will use the term *parallel branches* to refer to subplans that are unordered with respect to each other. Here we define the concept of parallel interaction; the following sections describe two plan critics that recognize and deal with these interactions. The goal phantomization critic takes advantage of helpful interactions so as not to produce inefficient plans, and the harmful interaction critic corrects harmful interactions that keep the plan from accomplishing its overall goal.

First, we will define helpful and harmful interactions in the context of classical planners. An *expected effect* is any formula the planner expects to be true within a subplan. In addition to the effects of actions, this includes any other condition the planner is monitoring and expecting to be true. For example, in SIPE, expected effects include the effects of precondition, phantom, choiceprocess, and goal nodes. If two branches of a plan are in parallel, an interaction is defined to occur when an expected effect in one branch (at any level in the hierarchy) possibly codesignates with an expected effect in another branch. If the effects agree in sign, the interaction may be helpful, otherwise it may be harmful. Since the actions in a SIPE plan explicitly list their expected effects (a feature shared by other classical planners), it is always possible to recognize such interactions. (In a hierarchical planner, however, they may not appear until lower hierarchical levels of both branches have been planned.) As we shall see below, the harmful interaction critic does not consider all effects of actions as expected effects — side effects are sometimes ignored.

The planner can sometimes take advantage of a situation in which a goal in one branch is made true in another branch (a helpful interaction). Suppose we solve the standard block-

world problem of getting A on B on C, starting with A and C on the table and with B on A. In solving the B-on-C parallel branch, the planner will plan to move B onto C, thus making A clear and C not clear. Now, when an attempt is made to move A onto B, the goal of making A clear becomes part of the plan. Since A is not clear in the initial state, the planner may decide to make it true by moving B from A to the table (after which it will have to move A onto B). In this case it would be better to recognize the helpful effect of making A clear, which happens in the parallel branch. Then the planner could decide to get B on C first, after which both A and B are clear and the (On A B) goal is easily accomplished.

The goal phantomization critic must decide whether or not to take advantage of a helpful interaction. Ordering the parallel branches sequentially is the best solution in our example because B must be put on C first in any case, but in other problems an ordering suggested to take advantage of helpful effects may prevent eventually achieving the overall goal. In addition to the decision of whether to do the ordering, there are options of choosing instantiations to take advantage of possibly codesignating helpful interactions. As described in the next section, SIPE uses heuristics to make such decisions without exploring this search space.

If an interaction is detected that makes an expected effect false in a parallel branch, there is a problematic (i.e., possibly harmful) interaction, which may mean that the plan is not a valid solution. The interaction will be harmful only if the effects necessarily codesignate. For example, suppose the planner does not act upon the helpful interaction in our problem and proceeds to plan to put B on the table and A on B in the (On A B) branch. The plan is no longer a valid solution (since B is being moved to both C and the table in parallel branches). The planner must recognize this by detecting the harmful interaction. Namely, the goal of having B clear in the B-on-C branch is made false in the A-on-B branch. The planner must then decide how to rectify this situation. SIPE's approach to this problem is part of the harmful interaction critic described below.

NOAH and NONLIN detected harmful parallel interactions by constructing a TOME (table of multiple effects). NOAH ignored helpful interactions, though NONLIN noticed them. While SIPE's critic for dealing with harmful interactions does a computation similar to that done in constructing a TOME, there are many enhancements that are described below.

9.3 Goal Phantomization

Goal phantomization is the process of "achieving" a goal by having it already be true at the point in the plan where the goal occurs. This is a trivial process in ground (instantiated) nonlinear plans: either a goal is true or it isn't. However, the introduction of either variables or nonlinearity complicates the situation, as they provide choices of methods for accomplishing a goal. The system can choose various instantiations or take advantage of helpful parallel interactions. (Goals in SIPE can be encoded as either choiceprocess nodes or goal nodes, and the following discussion applies to both.) In general, SIPE remains true to its least-commitment philosophy by refusing to instantiate variables or add ordering constraints unless it forced to do so. However, it is often necessary to achieve goals by means of these actions in order to produce optimal plans. In general, the planner cannot predict the correctness of applying these actions unless it completely investigates all the consequences of such a decision, which entails a combinatorial search. Since SIPE does not generate backtracking points for these actions, it provides several user-selectable options for phantomizing goals with these actions. These options have proved quite useful in domains that have been implemented.

Regarding the use of instantiation to accomplish goals, the system provides the user with several options. One choice is to instantiate whenever possible. This does not actually instantiate variables unless there is only one possible instantiation to accomplish the goal; in other cases it simply posts a pred constraint (which is the least commitment consistent with solving the goal). This is the choice most often used in the domains implemented in SIPE. Another choice is never to instantiate, which never posts constraints of any kind, thus avoiding any commitment. A third choice is to instantiate, but only when there is one possible binding that will accomplish the goal. This will post instan and same constraints but not pred constraints. In addition, the user can provide a list of action and predicate names that are exceptions to the general choice. If either of the first two choices above is selected, an excepted predicate/action is treated as if the other choice had been selected. When the third choice above is selected, no constraints will be posted on an exception. Exceptions were used in one domain, and were quite useful because the desirability of phantomizing by instantiation aligned itself nicely with predicate names.

Options for using helpful parallel interactions to accomplish goals are more restricted because adding ordering constraints is an irrevocable act and instantiations may also have to

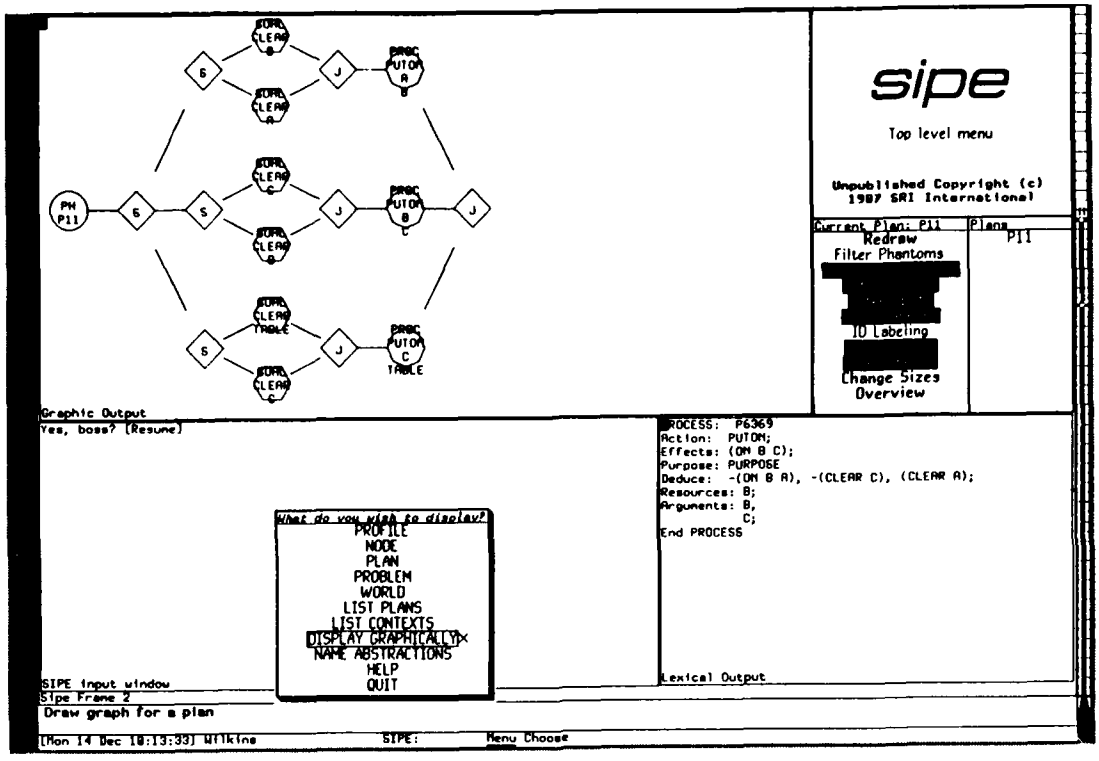


Figure 9.1: Plan for Three-Conjunct Block Problem

be made. SIPE provides three options to the user: 1) never add ordering constraints for the purpose of phantomization, 2) only add ordering constraints when no other constraints are required to accomplish the phantomization, and 3) add ordering constraints when there is a parallel branch with an effect that has only one possible instantiation for accomplishing the phantomization. The second option, no additional constraints, means that the helpful effect in a parallel branch is necessarily codesignating with the goal being phantomized, and that no possibly codesignating clobberers exist on this branch after this effect. In other words, the goal we are phantomizing is guaranteed to be true at the end of the parallel branch. The third choice also requires that no possibly codesignating clobberers exist, but allows the effect to be possibly codesignating as long as there is only one possible instantiation that will make it codesignate. Thus, addition of only same and instan constraints will guarantee the phantomization at the end of the parallel branch.

A further complication when adding ordering constraints is that several linearizations (i.e., additional ordering constraints) may be proposed at the same time. In this case, SIPE uses the plan rationale to reason about which should come first. Let us consider the standard block-world problem of getting A on B on C, using the Puton operator shown in Chapter 3. In this case, the initial goal is given as three parallel conjuncts, achieving (On C Table), (On A B), and (On C B). Assuming a initial state where the blocks are stacked on each other in reverse order, SIPE produces the plan shown in Figure 9-1 after one planning level. The goal phantomization critic, using either of the two options which allow linearizations, proposes to put C on the table before the other two branches, because it makes B clear which is a goal in both the other branches. It also proposes to put B on C first because it makes B clear, which is a goal in getting A on B.

While a simple heuristic could recommend that the former proposed linearization come first in this case, it is more general to reason about the rationale behind the branches. SIPE looks at the protect-until values of the goals being phantomized by the proposed linearizations, and takes these to be the reasons for the linearizations. Thus, the Clear B goal in the B-on-C branch is being protected until (On B C) is achieved, so the system assumes the reason for proposing this linearization is to achieve (On B C). SIPE delays any linearization that would accomplish the reason for doing a second proposed linearization. It would not be reasonable to consider the second linearization if its reason had already been achieved. Putting C on the table first does not achieve (On A B) which is the reason for the other linearization, but putting B on C first would achieve the reason for putting C on the table. This algorithm therefore first puts C on the table and then B on C.

The above options for phantomization through instantiation and linearization add flexibility to the system and provide more tools for doing efficient planning in particular domains. The choice of certain options may not guarantee completeness in a given domain. While it would be nice to have the planner solve the problem correctly without the user choosing these options, that involves solving more than one combinatorial problem. The set of tools described here permits solutions to problems that would be beyond the reach of a complete planning system. The heuristics involving the existence of only one possible instantiation are useful because one often does not want to make a commitment when the possibilities are rich, but does when it is the only choice. We are not committed in principle to the avoidance of backtracking points in goal phantomization, although this option has been selected to make

SIPE efficient. It may be desirable to allow choice points for the solving of goals by either instantiation or linearization in domains where the problem can be kept tractable.

NOAH was not able to take advantage of helpful interactions. It solved the block stacking problem by responding to harmful interactions; if the conjuncts had not interacted in a harmful way, NOAH would not have ordered them to take advantage of any helpful effects. NOAH did have an "eliminate redundant preconditions" critic that eliminated preconditions that occurred twice in the plan, but this could not recognize and react appropriately to a single precondition that was an integral part of the plan being achieved in a parallel branch. NONLIN, on the other hand, did have an ability to take advantage of helpful interactions, though it was not described in detail. This is an important ability in many real-world domains, since helpful side effects occur frequently. For example, if parallel actions in a robot world both require the same tool, only one branch need plan to get the tool out of the tool box.

9.4 Solving Harmful Interactions

As with helpful interactions, there is no easy way to solve harmful interactions. Here too a correct solution may require that all future consequences of an ordering decision be explored. Stratagems other than ordering may be necessary to solve the problem. For example, a new operator may need to be applied at a higher level. Consider the problem of switching the values of the two registers in a two-register machine. Applying the register-to-register move operator creates a harmful interaction that no ordering can solve, since a value is destroyed. The solution to this interaction involves applying a register-to-memory move operator at a high level in order to store one of the values temporarily. Correcting many types of harmful interactions efficiently seems very difficult in a domain-independent planner — domain specific heuristics may be required.

In NOAH, problematic parallel interactions were detected by the TOME and handled by the resolve-conflicts critic. SIPE has several techniques for dealing with this problem that greatly extend the capability in NOAH. The most important one is its ability to use resource reasoning to handle many problems that other classical planners would have to resolve through harmful parallel interactions. Since resource reasoning is more efficient and easier to express in operators, this transferal of effort is advantageous for several reasons. Resource reasoning is an important contribution of SIPE and is described in Chapter 10.

Another important technique for dealing with nonlinear actions is distinguishing between main effects and side effects of an action, as discussed in Chapter 5. SIPE only recognizes and resolves interactions dealing with the main effects of nodes. (One of the interacting effects may be a side effect, but interactions between two side effects are ignored). This greatly reduces the computational burden by not requiring the system to resolve conflicts that do not matter anyway. The user can use the system's flexibility to properly represent predicates as main or side effects. SIPE also simplifies the problem by not shuffling actions between two parallel branches — it will only order the actions by putting one branch before or after the others. Although this does prevent some elegant solutions from being found (e.g., the Sussman anomaly — see examples below), it retains efficiency while not being overly restrictive.

The parallel interaction critic makes use of the plan rationale in determining which solvers to apply to a harmful interaction. Suppose a particular predicate is made false by an effect on one parallel branch and true by a different effect on another parallel branch. Depending on the rationale for including these effects in the plan, it may be the case that each effect is not relevant to the plan (an extraneous side effect), or must be kept permanently true (the purpose of the plan), or must be kept only temporarily true (a precondition for later achievement of a purpose). SIPE's ability to specify plan rationale flexibly and to separate side effects from main effects enables it to distinguish these three cases accurately, something NOAH and its predecessors could not do.

Solutions to a harmful interaction may depend on which of these cases holds. Let us call the three cases side-effect, purpose, and precondition, respectively, and analyze the consequent possibilities. If the effect in conflict on one branch is a precondition, the proposed solution is to further order the plan, first doing the segment of the plan that extends from the precondition on through its corresponding purpose. Once this purpose has been accomplished, there will be no problem in negating the precondition later. This solution applies no matter which of the three cases applies to the other conflicting effect. (Thus if both conflicting effects are preconditions, two different solutions can be proposed.)

In the case of a side-effect that conflicts with a purpose, the proposed solution is to order the plan so that the side effect occurs before the purpose; thus, once the purpose has been accomplished it will remain true. When both conflicting predicates are purposes, there is no possible ordering that will achieve both purposes at the end of the plan. The planner

must backtrack and use a different operator at a higher level, or plan to reachieve one of the purposes later. The latter is attempted first by SIPE, and backtracking will happen if this fails. Reachieving a purpose is accomplished by calling the Insert-parallel replanning action described in Chapter 11. This replanning action checks which parallel postconditions on the join node are not true and inserts goal nodes for them after the join node (requiring that all parallel postconditions be true at the end of the newly inserted goals).

This briefly summarizes SIPE's algorithm for dealing with harmful interactions. It should be noted that none of the above proposed solutions can be guaranteed to produce the best (according to some metric, e.g., shortest) solution. Systems like NOAH and NONLIN do similar things with harmful interactions. However, SIPE provides methods for more precise and efficient detection, through its plan rationale and resource reasoning. It should be emphasized that many interactions that would be harmful in the other systems are dealt with in SIPE by the resource-reasoning mechanisms and therefore do not need to be analyzed. Unlike previous systems, SIPE ignores interactions among side effects.

9.5 Adding Ordering Constraints

The addition of ordering constraints is used to solve resource conflicts, to phantomize goals, and to solve harmful parallel interactions. SIPE collects proposed linearizations and attempts to optimally order them (as described in goal phantomization) before carrying them out. Once they are carried out, the interaction of the newly ordered part of a plan with the remainder of the plan is complex. In addition to possible interactions between various nodes in the plan, there may even be changes in the information contained in the nodes. In particular, deduced effects may be different, and optional-not-same constraints added by the system to avoid resource conflicts may no longer be valid. For this reason, SIPE generates the newly ordered plan and analyzes the problems caused by the new ordering. If they are extensive, the linearization is rejected.

To check a possible linearization, the system first removes optional-not-same constraints that are no longer valid and recalculates all the deduced effects in the remainder of the plan. It then checks for interactions between nodes in the plan. When SIPE was initially implemented, it had simple heuristics for accepting a newly ordered plan that often resulted in incorrect and nonoptimal plans. Once the system's execution monitoring and replanning capabilities were

implemented (see Chapter 11), it became obvious that the addition of ordering constraints could be treated exactly like an unexpected occurrence during execution. Checking the proposed linearization proceeds by calling the *problem recognizer* of the replanning module which finds any possible problem that the linearization might create in the remainder of the plan.

If problems are discovered, the linearization is generally rejected. In certain cases however, replanning actions are applied to further modify the plan after linearization, sometimes adding goals and sometimes removing subplans. New goals are inserted when the parallel postconditions of a join node are not true. If the linearization makes some goal true that actions later in the plan are trying to achieve, some part of the remaining plan can be removed and replaced by a phantom node. The replanning module already has actions for doing exactly this, and the example in the next section shows a trace of the system shortening a plan in this manner. This both corrects invalid plans and makes suboptimal plans more efficient. The power to modify plans in this way makes the system considerably more powerful than previous classical planners. Chapter 11 describes how the replanner uses the plan rationale to find problems in a plan and correct them.

9.6 Examples

Block-world problems have been used throughout to explain the operation of the critics because their simplicity allows the issues involved to be clearly brought out. Using the critics described in this chapter, SIPE correctly solves all the standard block-world problems involving three blocks. The only problem for which a nonoptimal solution is obtained is the Sussman anomaly (C is initially on A, and A and B are on the table) when the initial goal is given as the two parallel conjuncts (On A B) and (On B C). In this case, SIPE produces a plan in which B is first moved onto C and then moved back to the table. This shortcoming is not a failure of the critics, but rather a consequence of our assumption that parallel branches will not be shuffled together. Given the above restrictions on the representation of the problem, finding the optimal solution requires separating the goal of clearing A in the On A B branch from the rest of the branch.

SIPE does produce the optimal solution to the Sussman anomaly when given the three-conjunct problem (with (On C Table) as a conjunct), because the goal phantomization critic

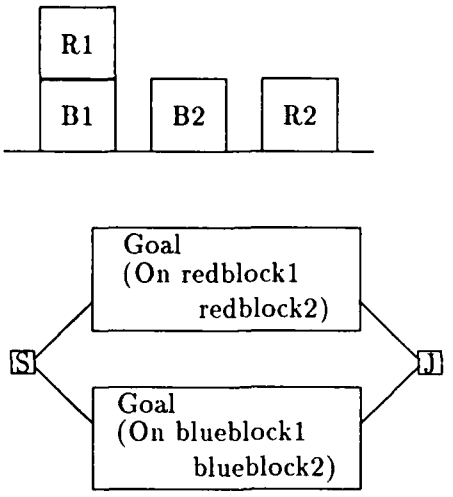


Figure 9.2: Initial Blocks World and Problem To Be Solved

correctly puts C on the table first, as described above. With the three-conjunct problem, the system produces optimal solutions for all possible problems from all possible initial states. Requiring three conjuncts in the problem is quite reasonable. People would have a hard time solving the Sussman anomaly with no more knowledge of the On relation than we give classical planners. The latter generally have no way of knowing that towers must be built from the bottom up, or even that blocks on the bottom of a tower cannot also be on top of the tower. Providing the information that C must be on the table is something all humans would have incorporated in their background knowledge of the world.

The simple block-world problem shown in Figure 9-2 has been useful in testing and developing SIPE. This assumes a world of red blocks and blue blocks, and the problem is to get any red block on top of another red block in parallel with getting any blue block on top of any other blue block. The problem has four variables and no constants, which makes it useful for testing system performance when nothing is instantiated. Many algorithms (from deducing effects to plan critics) which work on ground problems fail on this problem. Because of the use of variables and constraints, most previous classical planners could not have represented and solved this problem. An annotated trace of SIPE solving this problem is presented below. It shows the order and frequency with which the automatic search invokes various mechanisms, and also uses many of the critics described above, including the use of

replanning actions to shorten the plan after a linearization.

```
Planning at level 1.
2. unexpanded nodes: P4529 P4523
Expanding node P4529 with action PUTON
Expanding node P4523 with action PUTON

New plan produced:
PLANHEAD:  P4606
SPLIT:  C4605
   Parallel branch:
     SPLIT:  C4618
        Parallel branch:
          GOAL:  P4623 (CLEAR object1-N4526)
        Parallel branch:
          GOAL:  P4620 (CLEAR block1-N4524)
     JOIN:  C4619
     PROCESS:  P4626 PUTON.PRIM block1-N4524 object1-N4526
   Parallel branch:
     SPLIT:  C4595
        Parallel branch:
          GOAL:  P4600 (CLEAR object1-N4532)
        Parallel branch:
          GOAL:  P4597 (CLEAR block1-N4530)
     JOIN:  C4596
     PROCESS:  P4603 PUTON.PRIM block1-N4530 object1-N4532
JOIN:  C4607

Checking constraints. Constraints satisfiable.
Success: node P4623 changed to phantom.
Success: node P4600 changed to phantom.
Success: node P4597 changed to phantom.
Found 0. harmful interactions.
checking parallel postconditions:
  (ON redblock3-N4530 redblock4-N4532)(ON blueblock3-N4524 B2)
```

The above is the first planning level which applies the Puton operator to both branches of the problem, then applies the critics which do not find any problems. The three phantomizations instantiate one of the blueblock variables to B2 (because there is only one possible instantiation that make a blue block clear), and place pred constraints on the redblock variables.

```
Planning at level 2.
1. unexpanded nodes: P4620
Expanding node P4620 with action CLEARTOP
matching condition: (ON block1-N4638 object1-N4524)
(COLLECTED 1. POSSIBILITIES)
Adding INSTAN constraint: these two bound: block1 N4638, R1
Adding INSTAN constraint: these two bound: object1 N4524, B1
This phantom no longer true, changed to goal: P4674
```

```
This phantom no longer true, changed to goal: P4685

New plan produced:
PLANHEAD:  P4651
SPLIT:  C4650
   Parallel branch:
     SPLIT:  C4675
        Parallel branch:
          GOAL:  P4685 (CLEAR block1-N4530)
        Parallel branch:
          GOAL:  P4674 (CLEAR object1-N4532)
     JOIN:  C4676
     PROCESS:  P4677 PUTON.PRIM block1-N4530 object1-N4532
   Parallel branch:
     PROCESS:  P4642 PUTON R1 object2-N4636
     GOAL:  P4646 (CLEAR B1)
     PROCESS:  P4653 PUTON.PRIM B1 B2
JOIN:  C4654

Checking constraints. Constraints satisfiable.
Success: node P4685 changed to phantom.
Success: node P4674 changed to phantom.
Success: node P4646 changed to phantom.
Found 0. harmful interactions.
checking parallel postconditions: (ON redblock3-N4530 redblock4-N4532)(ON B1 B2)
```

The second planning level applies Cleartop to move R1 off of B1 to an unspecified place. These instantiations are forced because B1 is the only blue block other than B2, and the two blueblock variables have been constrained to not be the same as each other by the application of Puton at the first planning level. Phantoms that do not necessarily codesignate with the effects that make them phantoms are reset to goals before the critics are called. Again the critics find no problems as the redblock variables have not yet been instantiated.

```
Planning at level 3.
1. unexpanded nodes: P4642
Expanding node P4642 with action PUTON
This phantom no longer true, changed to goal: P4700
This phantom no longer true, changed to goal: P4723
This phantom no longer true, changed to goal: P4734

New plan produced:
PLANHEAD:  P4699
SPLIT:  C4698
   Parallel branch:
     SPLIT:  C4724
        Parallel branch:
          GOAL:  P4734 (CLEAR object1-N4532)
        Parallel branch:
          GOAL:  P4723 (CLEAR block1-N4530)
```

```
     JOIN:  C4725
     PROCESS:  P4726 PUTON.PRIM block1-N4530 object1-N4532
   Parallel branch:
     SPLIT:  C4686
        Parallel branch:
          GOAL:  P4691 (CLEAR object1-N4636)
        Parallel branch:
          GOAL:  P4688 (CLEAR R1)
     JOIN:  C4687
     PROCESS:  P4694 PUTON.PRIM R1 object1-N4636
     GOAL:  P4700 (CLEAR B1)
     PROCESS:  P4702 PUTON.PRIM B1 B2
JOIN:  C4703

Checking constraints. Constraints satisfiable.
Success: node P4734 changed to phantom.
Success: node P4723 changed to phantom.
Success: node P4691 changed to phantom.
Success: node P4688 changed to phantom.
Success: node P4700 changed to phantom.
Found 0. harmful interactions.
checking parallel postconditions: (ON redblock3-N4530 redblock4-N4532)(ON B1 B2)
```

The third planning level applies Puton to move R1 to some unspecified place. Again the critics find no problems as the redblock variables have not yet been instantiated.

```
Planning at level 4.
0. unexpanded nodes: success
recursion succeeds, applying critics
Checking constraints. Constraints satisfiable.
Adding INSTAN constraint: these two bound: object1 N4636, TABLE
Adding INSTAN constraint: these two bound: object1 N4532, R2
Adding INSTAN constraint: these two bound: block1 N4530, R1
Adding INSTAN constraint: these two bound: object3 N4727, B1
Adding INSTAN constraint: these two bound: object3 N4678, B1
Adding INSTAN constraint: these two bound: object3 N4610, B1
Adding INSTAN constraint: these two bound: object3 N4581, B1
```

The goal phantomization critic has phantomized all goals, so the planning is finished and the critics must be rechecked in light of the new constraints added by these phantomizations. Solving the global constraint network forces all variables to be instantiated, producing the invalid plan shown below.

```
Short version of final plan (use DISPLAY to view):
PLANHEAD:  P4699
SPLIT:  C4698
   Parallel branch:
     PROCESS:  P4726 PUTON.PRIM R1 R2
```

```
   Parallel branch:
     PROCESS:  P4694 PUTON.PRIM R1 TABLE
     PROCESS:  P4702 PUTON.PRIM B1 B2
JOIN:  C4703

Found 1. conflict.
RESOURCE R1 IN BRANCH BEGINNING WITH NODE C4724 conflicts with
      RESOURCE R1 IN BRANCH BEGINNING WITH NODE C4697

Ordering the plan.  Linearized part of plan to end:
PROCESS:  P4726 PUTON.PRIM R1 R2
PROCESS:  P4694 PUTON.PRIM R1 TABLE
PROCESS:  P4702 PUTON.PRIM B1 B2
```

Since the Puton operator describes the block it is moving as a resource, the resource-reasoning critic described in the next chapter recognizes the problem in this plan and fixes it by adding further ordering constraints. If the Puton operator had not declared a resource, the harmful interaction critic would have proposed the same linearization (though at a higher computational cost).

```
Calling problem recognizer.
Future precondition failed: (ON R1 B1)
Removing part of plan from net.  Deleted part starts with:
PRECONDITION:  P4696
Goals: (ON R1 B1);
Effects: (ON R1 B1);
Protect-until: (CLEAR B1);

Deleted part ends with:
PHANTOM:  P4700
Goals: (CLEAR B1);

Node to be copied and inserted:
GOAL:  P4620
Action:  CLEARTOP expansion: (P4641 . P4646) context: (TOP . TOP)
Goals: (CLEAR B1);
Effects: (CLEAR B1);
Protect-until: (ON B1 B2);
```

The solver for adding ordering constraints calls the problem recognizer which recognizes that the precondition for clearing B1 (R1 being on top of B1) is no longer true when expected. It the calls the Pop-redo replanning action (see Chapter 11) which removes the wedge of the plan that was created to clear B1 and replaces it with the goal (from the top of the wedge) of clearing B1 (which will become a phantom).

```
checking parallel postconditions: (ON R1 R2)(ON B1 B2)
Planning at level 4.
```

```
Success: node P4773 changed to phantom.

New plan produced:
PLANHEAD:  P4844
PROCESS:  P4846 PUTON.PRIM R1 R2
PROCESS:  P4852 PUTON.PRIM B1 B2

Found 0. harmful interactions.
checking parallel postconditions: (ON R1 R2)(ON B1 B2)

Planning at level 5.
0. unexpanded nodes: success
recursion succeeds, applying critics
Found 0. harmful interactions.
checking parallel postconditions: (ON R1 R2)(ON B1 B2)
SIPE solved problem, use DISPLAY to see plan.
Evaluation took 6.6017348 seconds of elapsed time.
```

The evaluation time was taken from a run in which the tracing was not on. The type of trace printed here takes several more seconds.

This example brings out many of the features of the automatic search and the plan critics. It shows the application of the critics after each planning level and their reapplication whenever the adding of constraints solves the problem. An invalid plan is produced, it is correctly linearized into a suboptimal plan, and the problem recognizer then modifies this plan to make it optimal. The reason SIPE's solution incorporates these steps is its least commitment strategy regarding phantomization by instantiation. The system keeps open different possibilities for the variables instead of instantiating them immediately. In a more complex problem, this may have enabled a more elegant solution to be found. For example, suppose some other problem constraint had, after several levels of planning, required that R2 be put on R1. SIPE could have incorporated this in its plan at any planning level, since the redblock variables were never instantiated. Then the parallel branch of putting R1 on the table and B1 on B2 could have been ordered first, following by putting R2 on R1, thus obtaining the optimal solution without any extraneous effort (such as retracting part of a plan). This example also provides a scale for measuring our claims of efficiency for SIPE. This problem, with its many applications of critics and solvers that require many unifications of uninstantiated variables that have pred constraints, takes less than 7 seconds on a Symbolics 3600.

Chapter 10

Resources: Reusable, Consumable, Temporal

When humans are doing planning and scheduling, they frequently describe tasks in terms of the resources that are required. This appears to be a very useful concept for reasoning about how different activities interact, and is used by many scheduling algorithms. When there is a harmful interaction between two actions in a nonlinear plan, there is often something that can be considered a resource for which the two actions are contending. The concept of a reusable resource, in particular, is ubiquitous. For example, whenever one process is using a tool (e.g., a hammer or soldering iron), no other process should plan to use the same tool simultaneously.

Thus it is natural to consider reasoning about resources in a planning system. The wide applicability of resource reasoning capabilities means the effort required to encode them should be rewarded. In this chapter, we describe SIPE's implementation of such capabilities, which represent the first attempt to incorporate them into a classical AI planning system. Three significant advantages have been obtained by reasoning about resources. First, it is a more natural and graceful way to interact with users, since humans are more comfortable talking about resource requirements and conflicts than about harmful parallel interactions. Second, the system obtains computational advantages because it is able to detect resource conflicts earlier and with less effort than by using the traditional methods of analyzing harmful parallel interactions. Third, with the implementation of consumable and temporal resources

which require numerical reasoning, the system has the power to express an important new class of domains.

Resources can be viewed as a powerful tool that can be employed by the user to represent domain-specific knowledge concerning the behavior of actions. There are many types of resources that might be useful, and SIPE provides two of the most general types of resources, reusable and consumable. The formalism for representing operators in SIPE includes a means of specifying that some of the variables associated with an action or goal actually serve as resources for that action or goal. In the Puton operator in Chapter 3, we saw the block being moved described as a resource. This causes the system to treat these objects as reusable resources, and no possibly parallel action will be permitted to use the same object as an argument and/or resource.

There is no simple way to describe the handling of a consumable resource, so SIPE provides mechanisms for declaring the production and consumption of resources, and calculates predicates describing their levels so that the user can specify goals using these predicates. This provides flexibility, as the user can place whatever requirements he desires on the levels of consumable resources, as long as they can be expressed with the provided predicates in the operator specification language of the system. Our implementation provides the basis for reasoning about producible and consumable resources, as well as limited forms of temporal reasoning (e.g., specifying constraints on the starting time of an action). The same representations and algorithms work for both these tasks because time is considered to be a type of consumable resource — namely, one that can be consumed but not produced, and whose consumption in the course of parallel tasks is nonadditive.

We will explain the system's use of resources by using simple block-world examples. This is not to say that the block world is best reasoned about in terms of resources. Rather, the block world provides simple examples that can be used to clearly explain the underlying mechanism. This use of resources also reiterates the view that the mechanisms provided by SIPE are simply tools that can be used to solve a problem. Thus the resource-reasoning tool can be used to correctly produce block-world plans, even if it is not the preferred way to approach the problem. Resources are not necessary for SIPE's block-world planning; the system would solve the same problems just as well without declaring any resources.

10.1 Reusable Resources

The idea behind our implementation of reusable resources is simple: If objects are declared as resources, then no possibly parallel action will be permitted to use the same object as an argument. Because this is sometimes too strong a restriction, SIPE also permits the specification of *shared resources*, whereby a resource in one branch can be an argument in a parallel branch, but not a resource. The logical extension of this would be to have predicates that specify sharing conditions, but this has not yet been implemented.

If a parallel branch does use a resource as an argument/resource, then this resource conflict will be detected by the resource critic that is used in conjunction with the other critics described in the previous chapter. If a resource conflict is found, it is treated much the same as a harmful parallel action, with the system applying solvers to correct the problem. Generally, the solution for a conflict is the addition of ordering constraints. The example in Chapter 9 shows the resource critic solving a problem in this manner.

SIPE uses a heuristic for solving resource-argument conflicts. Such an interaction occurs when a resource in one parallel branch is used as an argument in another parallel branch (as distinguished from a resource-resource conflict, in which the same object is used as a resource in two parallel branches). The heuristic is to order the branch using the object as a resource before the parallel branch using the same object as an argument. The assumption is that the action using the object as a resource will be more dynamic with respect to the object (e.g., changing its state or location), and the action using the object only as an argument will be more static with respect to the object. Consequently, the resource action is done first to ensure that the object will be in more of a stable configuration when later actions occur that employ it as an argument. The above argument may not be convincing, and certainly this heuristic is not guaranteed to be correct, but it is another tool provided by the system that has been proved useful in the four domains encoded in SIPE. By simply setting a flag, the user can prevent the employment of this heuristic if it is inappropriate for a particular domain.

The system does attempt to form plans that do not produce resource conflicts when it is possible to do so. Whenever parallel actions with resources are inserted into the plan, SIPE finds all potential resource conflicts and posts optional-not-same constraints whose satisfaction will prevent these conflicts from occurring. (It also removes these constraints

appropriately when ordering constraints are added.) The resource-allocation critic described in the last chapter attempts to instantiate variables so that these constraints are satisfied. For example, if a robot arm is used as a resource in block-moving operators, the system will try to use different robot arms (if they are available) on parallel branches, thus avoiding resource conflicts. If only one arm is available, it will be assigned to both parallel branches. This will then be recognized as a resource conflict and the resource critic will attempt to resolve the conflict by further ordering the plan. In this way many conflicts are averted by intelligent assignment of resources.

Handling reusable resources in this way provides several advantages. First, the representation is efficient and easy to use. Simply declaring something as a resource brings to bear considerable functionality. As an example, consider planning the actions of several carpenters who are sharing a set of tools. In SIPE, one merely specifies all tools as resources in its operators and the system will automatically avoid conflicts in their use. In other classical planners, the user would have to axiomatize the requirements on multiple use of a tool. For example, operators may need preconditions requiring tools to be available and actions that list as effects the fact that tools are not available. Conflicts between these effects and preconditions/goals would then be recognized by the normal mechanism for handling harmful parallel interactions.

In addition, there are computational advantages. Resources conflicts can be detected with less effort than harmful parallel interactions, because checking whether two variables (at least one of which has been declared a resource) pose a resource conflict is simply a manner of unifying them. The primary advantage, however, is that conflicts can be detected earlier in the planning process, often at a higher level of abstraction. The planner does not need to plan down to the level where availability effects and goals produce harmful interactions. The resource conflict can be recognized as soon as the resource is inserted in the plan.

The block-world example of achieving (On A B) and (On B C) as a conjunction shows how SIPE's resource reasoning can be used as a tool to achieve these advantages. While the domain of constructing objects in a machine shop is more suitable for use of reusable resources, using resources in the block world is instructive because of the comparison this allows with block-world solutions of other planners. In this example, we will use the Puton operator from Chapter 3 that describes the block being moved as a resource. Figure 10-1 depicts a plan that might be produced by NOAH or NONLIN (or by SIPE without making

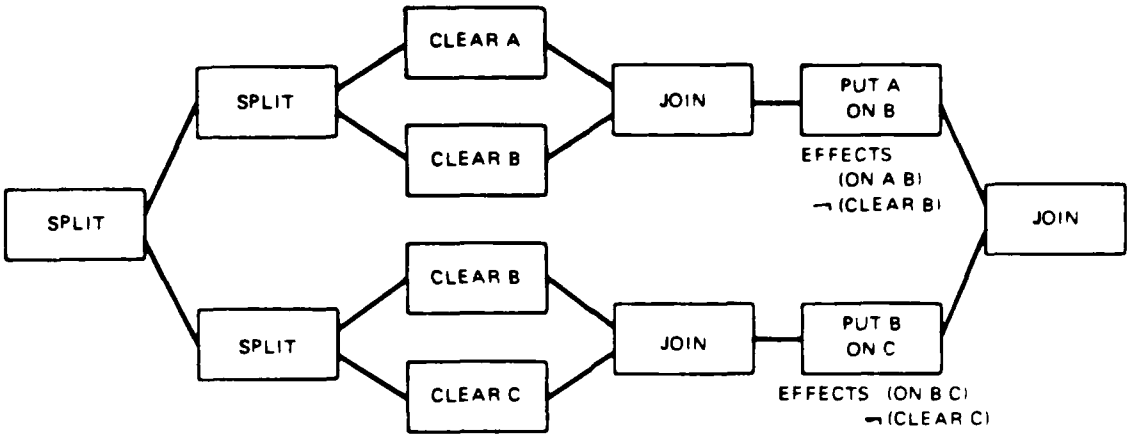


Figure 10.1: A Plan without Resources

use of resource reasoning) for this problem (from an arbitrary initial situation) after applying a standard puton operator to both original goal nodes. Figure 10-2 shows a plan from SIPE using the Puton operator with resources to expand the goal of getting B on C.

The central problem is to realize that B must be put on C before A is put on B (when starting from the Sussman anomaly, C must be moved before achieving either of these goals). Without resources, the condition of B being clear is eventually used to detect the conflict in Figure 10-1. NOAH and NONLIN both build up a table of multiple effects (TOME) that tabulates every predicate instance listed as an effect in the parallel expansions of the two goals. (SIPE would do a similar calculation, ignoring side effects, while calculating harmful interactions.) Using this table, the programs detect that B must be clear in the expansion of (On B C), but is made not clear in the other expansion. This problem is then solved by doing (On B C) first. Note that this interaction cannot be detected until the planning was proceeding to the abstraction level at which the Clear predicate is planned. In this example, that happens right away, but in a more complex domain it may not (e.g., in the robot domain, conflicts that are recognized at the level of the locational grid will not be detected until quite late in the planning process).

Using resources, SIPE can detect this problem and propose the solution (using its resource heuristic) without having to generate a TOME. In the Puton operator with resources, the

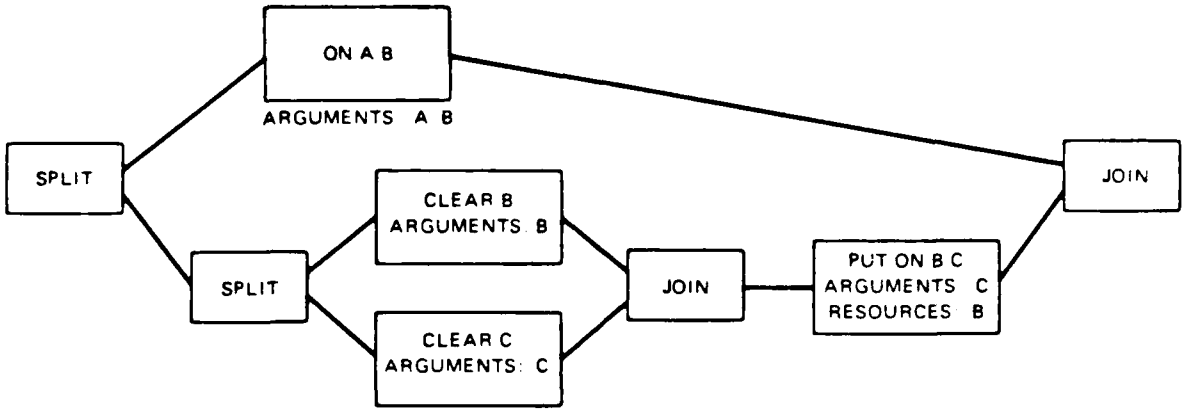


Figure 10.2: A Plan with Resources

block being moved is listed as a resource. Thus, as soon as the expansion of (On B C) with the Puton operator is accomplished and the plan in Figure 10-2 produced, SIPE recognizes that the plan is invalid because B is a resource in the plan for getting B on C and an argument in the (On A B) goal. SIPE's heuristic for solving a resource-argument conflict is to put the branch using the object as a resource first, so the resource critic proposes that B be put on C before A is put on B. This can de detected without analyzing interactions, and without expanding the (On A B) goal. Although the latter matters little in the block world, it could be very important if the goal would have to be expanded several levels before the interaction was detected.

This correction of the plan has the advantage of both being faster (interactions need not be analyzed) and providing earlier detection (which can avoid the search of a large part of the search space). While this may not be the best encoding of the block world, it does use SIPE's resource tool to good effect. This implementation does not do sophisticated reasoning about resources, such as analyzing the requirements on a particular class of objects and signalling a problem as soon as requirements exceed supply. This is a good area for later expansions of the system, although many such algorithms would appear to require domain-specific knowledge to be effective.

10.2 Representation of Numerical Quantities

SIPE has mechanisms for reasoning about numerical quantities, both continuous and discrete, which provides the basis for reasoning about producible and consumable resources. These mechanisms have been integrated within the existing framework of adding constraints to planning variables. The user invokes these mechanisms simply by declaring variables to be in the numerical class, and using constraints and certain distinguished predicates on these variables.

This design provides flexibility, as the user can place whatever requirements he desires on numerical quantities, as long as they can be expressed with the provided predicates in the operator specification language of the system. This provides a basis both for reasoning about producible and consumable resources and limited forms of temporal reasoning. The same representations and algorithms work for both these tasks because time is considered to be a type of consumable resource — namely, one that can be consumed but not produced, and whose consumption in the course of parallel tasks is nonadditive.

The system can employ all its standard planning algorithms to solve numerical problems by treating numbers as objects, and providing numerical variables, numerical constraints, and numerical predicates. The system automatically computes *level* predicates from predicates that describe the production and consumption of numerical quantities, and the user can specify goals and preconditions that involve these levels. First we describe the representations used for numerical objects, constraints, and predicates. Following that, we show how the representation is used to reason about consumable resources, then we describe how these representations are manipulated by the system.

• *Objects*

SIPE objects can include numbers, lists of numbers, ranges, and functions. These can be instantiations of planning variables or values of constraints posted by either the user or the system. Lists of numbers are provided for temporal reasoning along the lines of Ladkin's TUS syntax [16]. Initially, SIPE will add lists of numbers by using the list (60 24 31 12 0) as the limits for carrying. This is intended to correspond to minutes, hours, days, months, and years. The user can customize his temporal representation simply by setting a global variable to some other list. For example, one could use (31 12 0) when only concerned about

days, or (60 24) when planning the events of a single day. Lists of numbers have the same status as numbers and can occur anywhere a number is required (e.g., as the lower bound of a range, the value returned by a function, or the instantiation of a variable). Ranges are pairs of numbers that represent the lower and upper bounds of a range. Functions are represented by the function name and a list of arguments that can be any objects or planning variables known to the system. The function must return a number whenever it is called with completely instantiated arguments. The system ensures such instantiations before calling the function.

- *Variables*

Variables can be constrained to be members of two distinguished classes, *numerical* and *continuous*. The first is used for variables that will eventually be instantiated to one particular number, just as an ordinary variable is instantiated to an object. For example, this would be used to represent the starting time of a planned action. Variables in the continuous class will have values that vary with time and must be computed by the system. These can be used to represent the level of a consumable resources, e.g., the amount of petrol in a fuel tank. To finalize a plan, numerical variables should be instantiated to numbers, while continuous variables merely need to have satisfiable constraints. In addition, there will be phantom and precondition nodes in the plan whose truth the system must maintain to assure that the continuous variables have the right value at particular times. In the rest of this chapter, the term *numerical variable* will be used to refer to members of either of the above classes. In fact, the continuous class is implemented in the SIPE sort hierarchy as a subclass of the numerical class.

- *Constraints*

There are five constraints that may be posted only on numerical variables. (Other constraints may also be posted on numerical variables.) Most of their description in Chapter 4 is reproduced here.

Current value -- A numerical variable can be constrained to be the current value of a continuous variable at some point in the plan. This permits operators to reason about and place constraints on the value that some continuous variable has at some particular point in time.

Previous value — This is the same as current value except that the value is taken just before the current node instead of just after it.

Range — A variable can be constrained to lie within a certain range.

Function — A variable can be constrained to be the value of a certain function applied to any number of arguments. If some of these arguments are not instantiated, SIPE will compute a range from the function constraint by calling the function on all the possible instantiations.

Summary range — Since computing the above constraints (especially function constraints) can be expensive, it was necessary to address the problem of choosing between storage and recomputation. Our solution is to store the results of computing a noncontinuous variable's constraints by placing a summary-range constraint on the variable. (This cannot be done for continuous variables because their values vary with time.) These constraints are not posted by users, but only by the system.

• *Predicates*

Several distinguished predicates are treated specially by SIPE. These are *level, produce, and consume*. We will speak only of a *level* predicate, although there are actually five predicates which implement equality as well as relational predicates on levels: *level*, *level* <, *level* >, *level* <=, and *level* >=. These predicates occur at certain points in a plan and operate on the values of continuous variables at those points. All of these predicates can be viewed as having two arguments — one specifying the quantity being compared (e.g., the resource pool or fuel depot), the other specifying the numerical value that is being produced or consumed, or that must be equal to the value of the specified quantity. The former can, in fact, be a tuple of normal SIPE arguments (i.e., variables and objects known to the system). The latter is computed by taking into account the starting level and all nodes that are part of the plan previous to the current node whose effects contain *level, produce, and consume* predicates. All five *level* predicates can be used in preconditions and goals.

10.3 Consumable Resources

Before describing the algorithms used to manipulate the above representation, we will explicate our implementation and show its utility by describing some problems solved by SIPE. Consumable resources are implemented through the use of *level, produce, and consume* predicates and appropriate numerical variables. The clearest way to show this is to extend the standard blocks world. To reason about consumable resources, we assume that different blocks have different sizes and that block-moving actions use up the robot's fuel as a function of block size. Different moving actions have different functions for fuel consumption (e.g., Government-Puton uses fuel twice as fast as Fuel-Puton). SIPE was able not only to represent this world, but also to generate plans that required both selection of the correct operator (Fuel-Puton) and selection of smaller blocks so as to achieve fuel goals.

The Fuel-Puton operator, shown in 10-3, consumes the same number of units of fuel as the size of the block being moved. A simple problem in this domain is also shown (both are given in the input syntax accepted by SIPE). The problem requires the robot to get some block other than A on top of B, and afterwards have at least 5 but not more than 50 units of fuel remaining. This operator has three arguments in addition to those used in Puton (in the standard blocks world). These are the robot, a numerical variable for calculating the amount of fuel used by this operator, and a continuous variable used in the precondition to assure that the continuously varying fuel level of the robot is sufficient for this operator. Since the domain only requires one numerical level to be associated with the robot, the fuel level can be represented by the predicate (*Level robot*1 *numerical*1). If the robot had other levels, another argument could be added to the predicate, since SIPE permits a tuple of arguments to represent the quantity with which a numerical value is associated. (Any predicate with a different number of arguments refers to a different quantity). The precondition of Fuel-Puton assures that the robot has enough fuel to accomplish the actions of this operator (i.e., enough fuel to move *block*1).

In the initial world, the level of fuel can be specified with a *level* predicate. As actions are performed they will use and replenish fuel. An action that uses fuel simply specifies, in its effects, a *consume* predicate that denotes which fuel tank is being used (by giving the robot as the first argument), and the amount that is consumed (this second argument may be a function, a range, a number, or a list of numbers). Thus, the Fuel-Puton operator has a

Operator: Fuel-Puton
Arguments: block1, object1 Is Not block1, robot1,
numerical1 Is (Size block1), continuous1 Is (Size block1);
Purpose: (On block1 object1);
Precondition: (*Level* > robot1 continuous1);
Plot:
Parallel
Branch 1:
Goals: (Clear object1);
Branch 2:
Goals: (Clear block1);
End Parallel

Process
Action: Puton.Primitive;
Arguments: block1,object1,robot1,numerical1;
Resources: block1;
Effects: (On block1 object1),
(Consume robot1 numerical1);
End Plot End Operator

Problem: Prob1
Goal
Arguments: block1 Is Not A, B;
Goals: (On block1 B);
Goal
Arguments: robot1, continuous1 Is [5 50];
Goals: (*Level* > robot1 continuous1);
End Problem

Figure 10.3: SIPE Block World Operator and Problem Using Consumable Resources

consume predicate in the effects of the puton action in its plot. The system will then take this consumption into account in all queries about the fuel level, which may involve determining ranges within which the consumption must lie if the predicate has uninstantiated variables as arguments. Similarly, an action that replenished the fuel would post a *produce* predicate as an effect. Alternatively, an action that always filled the tank to some known level could simply post a *level* predicate as an effect. Such a *level* specification overrides previous produce and consume predicates, in the sense that SIPE will no longer look at them to calculate the fuel level, which is now given by the *level* predicate.

In the initial world of our example problem, Flakey has 40 units of fuel and A is on B. SIPE is told to first try Government-Puton (when achieving On goals), which consumes twice the number of units of fuel as the size of the block being moved. Since A has size 15, it will thus require 30 units of fuel using this operator to clear B. Since the smallest block has size 5, the goal of having 5 units of fuel left cannot be accomplished by using Government-Puton for both block moves. To meet the fuel goal, the system must therefore backtrack to use the Fuel-Puton operator and it must also select one of the smaller blocks to be moved onto B. An abbreviated annotated trace of SIPE solving this problem is presented below. It shows the interaction of numerical reasoning with the rest of the system.

```
Checking for phantom: P4351 (LEVEL> Flakey Continuous1-N4352)
Success: node P4351 changed to phantom.
Planning at level 1.
Finding applicable operators for node P4347
Operators found: (Government-Puton Fuel-Puton Time-Puton)
Adding INSTAN constraint: these two bound: Robot1 N4792, Flakey

New plan produced:
PLANHEAD:  P4811
SPLIT:  C4800
   Parallel branch:
     GOAL:  P4805 (CLEAR B)
   Parallel branch:
     GOAL:  P4802 (CLEAR Block1-N4348)
JOIN:  C4801
PROCESS:  P4808 Gov.Puton.Prim Block1-N4348 B Flakey Numerical1-N4794

Checking constraints.
Checking LEVEL phantom: P4812 (LEVEL> Flakey Continuous1-N4352)
Constraints satisfiable.

Checking for phantom: P4802 (CLEAR Block1-N4348)
Adding PRED constraint for predicate: CLEAR Length: 3.
```

```
Success: node P4802 changed to phantom.
Found 0. conflicts.
```

The plan after the first planning level is to move some block onto B. The level goal for robot fuel in the original problem is changed to a phantom since it is true in the current plan. Note that continuous variables cannot have pred constraints posted on them that guarantee that predicates will be true at particular times (because their values change over time). Thus, the constraint satisfaction critic checks this level phantom as part of checking constraints. (This effectively checks what would be pred constraints on continuous variables.)

```
Planning at level 2.
Finding applicable operators for node P4805
Operators found: (Cleartop)
Adding INSTAN constraint: these two bound: Block1 N4835, A

New plan produced:
PLANHEAD:  P4849
SPLIT:  C4846
   Parallel branch:
     GOAL:  P4871 (CLEAR Block1-N4348)
   Parallel branch:
     CHOICEPROCESS:  P4839 Government-Puton Fuel-Puton A Object2-N4833
     GOAL:  P4843 (CLEAR B)
JOIN:  C4850
PROCESS:  P4851 Gov.Puton.Prim Block1-N4348 B Flakey Numerical1-N4794

Checking constraints.
Checking LEVEL phantom: P4852 (LEVEL> Flakey Continuous1-N4352)
Constraints satisfiable.

Checking for phantom: P4871 (CLEAR Block1-N4348)
Success: node P4871 changed to phantom.
Checking for phantom: P4843 (CLEAR B)
Success: node P4843 changed to phantom.
Found 0. conflicts.
```

The plan after the second planning level is to move A off of B with either puton operator and to then move some block onto B with Government-Puton. This can be a valid plan if Fuel-Puton is used to move A, so all constraints are satisfiable.

```
Planning at level 3.
Expanding node P4839 with action Government-Puton
Adding INSTAN constraint: these two bound: Robot1 N4883, Flakey

New plan produced:
PLANHEAD:  P4906
SPLIT:  C4903
```

```
   Parallel branch:
     GOAL:  P4931 (CLEAR Block1-N4348)
   Parallel branch:
     SPLIT:  C4891
        Parallel branch:
          GOAL:  P4896 (CLEAR Object1-N4833)
        Parallel branch:
          GOAL:  P4893 (CLEAR A)
     JOIN:  C4892
     PROCESS:  P4899 Gov.Puton.Prim A Object1-N4833 Flakey 30.
     GOAL:  P4907 (CLEAR B)
JOIN:  C4908
PROCESS:  P4909 Gov.Puton.Prim Block1-N4348 B Flakey Numerical1-N4794

Checking constraints.
Checking LEVEL phantom: P4910 (LEVEL> Flakey Continuous1-N4352)
LEVEL phantom fails. Numerical constraints unsatisfiable.
can't allocate to satisfy constraints
```

Moving A off B is done with Government-Puton which uses 30 units of fuel. The block to put on B has not been chosen, but SIPE has computed a range for the fuel consumption based on the possible instantiations for this planning variable. The computation of level predicates takes this range into account and the system realizes the fuel goal can no longer be achieved, and backtracking is initiated.

```
Backtrack to try Fuel-Puton
Expanding node P4839 with action Fuel-Puton
Adding INSTAN constraint: these two bound: Robot1 N4937, Flakey

New plan produced:
PLANHEAD:  P4906
SPLIT:  C4903
   Parallel branch:
     GOAL:  P4979 (CLEAR Block1-N4348)
   Parallel branch:
     GOAL:  P4931 (CLEAR Block1-N4348)
   Parallel branch:
     SPLIT:  C4945
        Parallel branch:
          GOAL:  P4950 (CLEAR Object1-N4833)
        Parallel branch:
          GOAL:  P4947 (CLEAR A)
     JOIN:  C4946
     PROCESS:  P4953 Fuel-Puton.Prim A Object1-N4833 Flakey 15.
     GOAL:  P4956 (CLEAR B)
JOIN:  C4908
PROCESS:  P4957 Gov.Puton.Prim Block1-N4348 B Flakey Numerical1-N4794

Checking constraints.
```

```
Constraints satisfiable.

Checking for phantom: P4979 (CLEAR Block1-N4348)
Success: node P4979 changed to phantom.
Checking for phantom: P4931 (CLEAR Block1-N4348)
Success: node P4931 changed to phantom.
Checking for phantom: P4950 (CLEAR Object1-N4833)
Adding PRED constraint for predicate: CLEAR Length: 4.
Success: node P4950 changed to phantom.
Checking for phantom: P4947 (CLEAR A)
Success: node P4947 changed to phantom.
Checking for phantom: P4956 (CLEAR B)
Success: node P4956 changed to phantom.
Found 0. conflicts.
```

The planning succeeds at the fourth level because the fuel goal can be met as long as block D (which is size 5) is chosen for moving onto B. The critics notice this, make the instantiation, and produce the final plan.

```
Planning at level 4.
recursion succeeds, applying critics
Checking constraints.
Constraints satisfiable.
Adding SAME constraint: these two bound: Numerical1 N4794, Num-alloc N4993
Adding INSTAN constraint: these two bound: Block1 N4348, D
Adding INSTAN constraint: these two bound: Object1 N4833, TABLE

Short version of final plan (use DISPLAY to view):
PLANHEAD:  P4906
PROCESS:  P4953 Fuel-Puton.Prim A TABLE Flakey 15.
PROCESS:  P4957 Gov.Puton.Prim D B Flakey 10.

Found 0. conflicts.
recursion succeeded
SIPE solved problem, use DISPLAY to see plan.
Evaluation took 2.62 seconds of elapsed time
```

The evaluation time was taken from a run in which the tracing was not on. The type of trace printed here takes several more seconds.

This example shows how numerical reasoning is incorporated with the constraint-posting framework of SIPE. The system automatically computes level predicates, and backtracks appropriately when they fail. It is also able to choose instantiations for planning variables that will make level predicates true, and reason about the range a value must lie in when variables are not instantiated. How these capabilities are implemented is described in the next section. This example again provides a scale for measuring our claims of efficiency.

This problem, with its backtracking and numerical reasoning, takes less than 3 seconds on a Symbolics 3600.

10.4 Temporal Reasoning

SIPE's temporal reasoning capability is in the early stages of development, and has only recently been added to the system. While there has not been time to fully implement our design, it is still possible to solve interesting problems that go beyond what previous classical planners can do, as the following example illustrates. To test temporal reasoning, we assume that the time required to move a block is a function of its size. SIPE's representation allows encoding of a Wait operator, which allows the system to meet temporal goals by simply waiting for a certain length of time. Given problems that require actions to take place within prescribed time windows (e.g., an action could have a precondition stipulating that the time must be within a certain range), SIPE is able to pick blocks and operators correctly, producing a valid plan by inserting Wait actions. A trace would be similar to the one given for the fuel problem.

At present, the user has to make proper use of *level, produce, and consume* predicates with numerical variables and constraints in order to accomplish whatever temporal reasoning his domain requires. There are designated subclasses of Numerical and Continuous (Time and Clock). Variables in these classes will be treated as though they represent time, which means levels will be calculated in a nonadditive manner over parallel actions. In all other respects, temporal quantities are just like any other numerical quantities. Temporal values will generally be lists of numbers instead of integers, to take advantage of the TUS-like syntax. The starting time of an action can be given by a *level* predicate. Absolute times can also be represented by *level* predicates. Durations of actions can be specified by *consume* predicates. In the real world, one cannot generally produce time, but SIPE does not prevent the user from so using consumable resources.

We have not yet implemented some of our designs for temporal reasoning. It would be fairly straightforward to implement a much more useful syntax for the types of temporal reasoning most often encountered. For example, actions could have **start time** and **end time** slots that contained variables with numerical constraints on them. It would be easy to convert these to goal nodes or precondition nodes containing the appropriate *level* predicates to ensure

that the constraints on these slots would be met and maintained. Similarly, **duration** slots could easily be converted to appropriate *consume* predicates and be used to compute ending times from starting times. Such "syntactic sugar" would make the system much easier to use. In addition, it would be useful to build defaults for temporal values into the system. For example, the system might assume that one action would start immediately after the preceding one ended unless constraints on its starting time indicated otherwise.

Most of the algorithms we have designed for reasoning about time during parallel concurrent actions have not yet been implemented. For example, the start and end times of actions (i.e., the values of *level* predicates on temporal variables in the effects of actions) could be interpreted as ordering actions with respect to other actions. All the functions in SIPE that traverse a plan (e.g., plan critics, the truth criterion) could then utilize this ordering information to eliminate some possible orderings of parallel branches. While it is clear how this could be done, lack of time has postponed implementation.

10.5 Manipulating Numerical Quantities

In the fuel example, we saw the system using the representation previously described for numerical quantities. In this section, we describe the algorithms that perform these calculations. The unification algorithm must take numerical constraints into account, the truth criterion must handle level predicates specially, and the plan critics must check for problems with numerical variables. As one would expect, variables and nonlinearity again introduce complications for which SIPE provides heuristics.

Unifying two variables with most of the numerical constraints described in this chapter is straightforward. There are two complications: function constraints on variables, and the inefficiency of constantly recomputing the numerical range implied by all the numerical constraints on a variable. When uninstantiated numerical variables have function constraints, SIPE computes a range for the value of the variable by calling the function on all the possible instantiations that are currently consistent. In large domains, this may have to be replaced with some estimate that is easier to compute.

We have already stated that summary-range constraints are used to avoid recomputation of numerical quantities. These constraints summarize in one numerical range all the

consequences of the other constraints on a numerical variable. However, there remains the question of deciding when to recompute a summary-range which will change as planning progresses and more constraints are added and more instantiations made. In the case of adding constraints, we solve this by recomputing the summary-range of a variable when constraints are added to the latter. This is not done every time a constraint is added, but rather, every time a set of constraints is added. For example, when matching a precondition, the system waits until all the numerical constraints from the match are added and then recomputes all the numerical variables in the precondition before proceeding. In the case of instantiating variables, the problem is harder. Our solution is to include in the summary-range constraint not only the current value of the allowable range, but also a list of variables upon which this computed result depends. If any of these variables are instantiated, the summary-range is recomputed. This is done once every planning level by the plan critics that check resource allocation and constraint satisfaction.

We have already described the responsibilities of the plan critics regarding numerical variables. They must recompute summary-range constraints appropriately. They must also guarantee that predicates having continuous variables as arguments will be true at particular times. Because these values change over time, the system cannot post pred constraints on the variables. The constraint satisfaction critic therefore checks all phantom and precondition nodes in the plan that contain level predicates to ensure that the requirements on the continuous variables are still being met.

The truth criterion must determine the truth of a *level* query predicate with a continuous variable as an argument where the quantity corresponding to this variable had been produced and consumed over time. The algorithm is simple for linear plans whose nonnumerical variables are all instantiated. A range is computed within which the value of the continuous variable must lie. As the truth criterion regresses over actions with *produce* and *consume* effects that necessarily codesignate with the quantity being calculated, it updates the range being computed to incorporate the production/consumption (which may also be given as a range). If the regression reaches a node with an effect that is a *level* predicate for the quantity being calculated, then the regression can terminate as this predicate summarizes all previous production and consumption of this quantity.

As with the nonnumerical truth criterion, variables and nonlinearity both introduce problems. As with summary-range constraints, there is both a store versus recompute dilemma

and a problem with recomputation after further planning renders old computations obsolete. To solve these problems, the system stores the results of computing a level simply by posting a *level* predicate, with the value of the computation as an argument, on the deduced-effects list of the plan node from which the level was computed. This causes the system to behave exactly as we want without the need to continuously recompute levels, since the truth criterion will not regress past this predicate. Because these specially added *level* predicates are listed as deduced effects, they are not copied down to a lower planning level, thus affecting their necessary recomputation after the plan has been further specified.

The introduction of variables means that the level/produce/consume effect being processed by the truth criterion may only possibly codesignate with the query predicate (i.e., the tuple of nonnumerical arguments possibly codesignate with the corresponding tuple in the query predicate). Note that the list of possible establishers for the query predicate will contain *level, produce, and consume* effects. There is a search space that could be explored by the truth criterion: investigating possible codesignation constraints that would allow certain produce/consume effects to match the query predicate in order to make it true. SIPE does not search this space. Instead it divides the list of possible establishers into equivalence classes of predicates that must codesignate. When a new possible establisher is added, it is grouped with others that necessarily codesignate with it. After the regression has terminated, the system then computes a range for the continuous value from each equivalence class (combining the productions and consumptions in the class). Thus one *level* predicate that is a possible establisher is computed from each equivalence class. Note that there are no clobberers that accumulate: *level, produce, and consume* predicates are always unnegated. Whether a possible match is an establisher or clobberer depends upon how the numerical values in the query and possible match compare.

The effect of this heuristic is that SIPE will assume that *level, produce, and consume* predicates describe different quantities whenever possible. Thus it is the responsibility of the user to assure that the necessary same constraints are posted when these predicates describe the same quantity. Basically, abstract operators must encode whether or not different resource consumption actions at a lower abstraction level are consuming the same resource or not. The resource can still be an uninstantiated variable, but the same variable should be used in the two different actions if they are consuming the same resource. In our problem domains, this has been easy to do. This heuristic has proven quite useful while avoiding a combinatorial

search.

The last problem is the computation of levels after parallel actions. Our algorithm is simple and efficient, but not very powerful. It does not reason about different possible orderings of the parallel actions. It merely computes the minimum and maximum that could exist after the parallel actions, then uses this range as the result.

10.6 Summary

Resources are a powerful tool that can be employed by the user to represent domain-specific knowledge concerning the behavior of actions. SIPE provides tools for expressing and using both reusable and consumable resources within the classical planning paradigm. These tools include the ability to reason about numerical quantities, that allow representation of a significant new class of problems. No previous classical planners have used resources or numerical quantities. While the system's ability to express temporal and numerical problems does not approach that of systems designed for that purpose, it does provide new capabilities that retain the advantages of classical planners. This results in a more natural and graceful interaction with users and more efficient problem solving, in addition to the power to express an important new class of domains.

Chapter 11

Replanning During Execution

In real-world domains such as controlling a mobile robot, things do not always proceed as planned. Therefore, it is necessary to monitor the execution of a plan and to replan when things do not go as expected. In complex domains, it becomes increasingly important to use as much as possible of the old plan, rather than to start all over when things go wrong. The problem is the following: Given a plan, a world description, and some appropriate description of an unanticipated situation that occurs during execution of the plan, our task is to transform the plan, retaining as much of the old plan as is reasonable, into one that will still accomplish the original goal from the current situation. This process can be divided into four steps: (1) discovering or inputing information about the current situation; (2) determining the problems this causes in the plan, if any, (similarly, determining shortcuts that could be taken in the plan after unexpected but helpful events); (3) creating "fixes" that change the old plan, possibly by deleting part of it and inserting some newly created subplan; and (4) determining whether any changes effected by such fixes will conflict with remaining parts of the old plan.

The first step involves the challenging task of determining how to generate correct predicates from information provided by existing sensors (e.g., the pixels from the camera or the range information from ultrasound). This difficult problem, which is the subject of several other volumes, is of crucial importance to endowing a robot with a high-level planning capability. However, it is beyond the scope of our discussion of planning systems. Here, we concentrate on what a planning system might do with the predicates that are returned from the world (also a necessary part of the overall solution). We assume that new information

given to the planning system in the form of predicates it understands. The final three steps all involve determining which aspects of a situation later parts of the plan depend upon, and which effects listed in the original plan are still true (and which new ones should be added). The latter problem is an instance of the standard truth-maintenance problem.

In many domains, it may often be important to expend considerable effort in checking for things that might have gone wrong besides the unexpected occurrence already noticed. (Perhaps it is just the tip of the iceberg.) There is a substantial tradeoff involved here, as interpreting the visual input of unanticipated scenes may be expensive. However, we do not examine this problem either. In line with the classical planning assumption of a perfect world description, we assume that nothing has gone wrong besides reported errors and effects that can be deduced from them. The problem of uncertain or unreliable sensors or information is also largely unaddressed (although SIPE can specify that some predicates and variables are unknown).

The issues involved in solving these problems will become apparent as SIPE's replanning capability is described. While SIPE is not able to monitor the world directly, it can replan after it has been provided with arbitrary descriptions of the world in its own language. In many cases, it is able to retain most of the original plan by making some modifications, and is also capable of shortening the original plan when serendipitous events occur. This capability significantly extends those of previous classical planning systems by exploiting the rich structure in the system's plan representation and integrating the replanner within the planning system itself. This integration provides a number of benefits, of which the most important follow: the replanner uses the efficient truth criterion to discover problems and potential fixes quickly; the deductive causal theory is used to provide a reasonable solution to the truth maintenance problem described above; and the planner can be called as a subroutine to solve problems after the replanning module has inserted new goals into the plan. The replanner does use serendipitous effects to shorten the original plan in certain cases, and effectively eliminates the fourth step above as a problem by generating only those "fixes" that are guaranteed to work.

In general, optimal recovery from an arbitrary error poses an intractable problem. Often very little of the existing plan can be reused. One can always fall back on solving the original problem in the new situation, ignoring the plan that was being executed. Since the problem is so difficult, one would not expect very impressive performance, in terms of producing

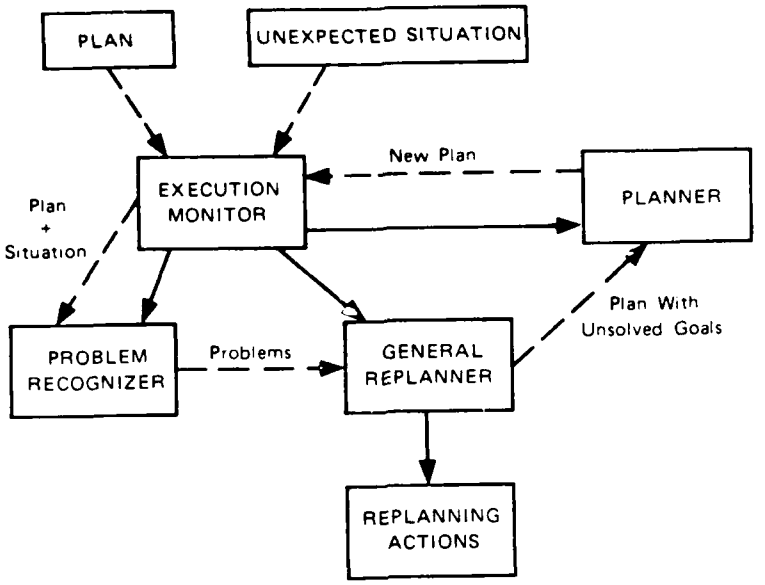


Figure 11.1: Control and Data Flow in SIPE's Replanner

optimal plans that reuse the original plan, from a domain-independent replanner. Producing more optimal plans requires domain-specific information for dealing with errors. In many domains, the types of errors that are commonly encountered can be predicted (e.g., the robot arm dropping something it was holding, or missing something it was trying to grasp).

The replanning part of SIPE tries to change the old plan, using heuristics to retain as much of it as possible in certain situations. An important contribution is the development of a general set of replanning actions that are used to modify plans. These are used both in the replanner and in the plan critics. The also have the potential for facilitating the addition of domain-specific knowledge about error recovery, since the user could specify which replanning actions to take in response to certain anticipated errors.

11.1 Overview of SIPE's Execution-Monitoring System

Figure 11-1 shows the various modules in the SIPE execution-monitoring system. The solid arrows show which modules call which others. The broken arrows show the flow of data and information through the system as it replans for an unexpected situation. These arrows are labeled with a description of the data being passed.

During execution of a plan in SIPE, some person or computer system monitoring the execution can specify what actions have been performed and what changes have occurred in the domain being modeled. The system changes its original world model permanently, so as to reflect the effects of actions already performed. At any point during execution, the *execution monitor* will accept two types of information about the domain: an arbitrary predicate whose arguments are ground instances, that is now true, false, or unknown; and a local variable name that is now unknown. SIPE first checks whether the truth-values for the new predicates differ from its expectations, and, if they do, it applies its deductive causal theory to deduce more changed predicates.

Once the description of the unexpected situation has been accumulated, the execution monitor calls the *problem recognizer*, which returns a list of all the problems it detects in the plan. The *general replanner* is then given the list of problems found by the problem recognizer and tries certain replanning actions in various cases, but will not always find a solution. The general replanner changes the plan so that it will look like an unsolved problem to the standard planner in SIPE (e.g., by inserting new goals). After the replanner has dealt with all the problems that were found, the planner is called on the plan (which now includes unsolved goals). If it produces a new plan, this new plan should solve correctly all the problems that were found and is given to the execution monitor.

11.2 Unknowns

Unknowns are not present in most classical planners, which generally assume complete knowledge of the world. Having unknown quantities requires fundamental changes down to the level of the truth criterion. If the truth-values of critical predicates are unknown, SIPE will quickly fail. None of the operators will be applicable, since neither a negated nor an unnegated predicate in a precondition will match an unknown predicate (i.e., one with a truth-value of unknown). Operators can require predicates to be unknown as part of their precondition, which is useful when there are appropriate actions to take in uncertain situations. A SIPE operator might produce a plan with an action to perceive the unknown value, followed by a conditional plan that specifies the correct course of action for each possible outcome of the perception action. The deductive causal theory can deduce unknown predicates.

The ability to specify variables as unknown is simply a tool provided by the system that

will presumably be useful in some domains, particularly in a mobile robot domain. The idea behind this tool is that the location of an object may become unknown during execution. Rather than make predicates unknown, which may cause the application of operators to fail, we simply say that the variable representing the location is instantiated to the atom *unknown*, rather than to its original location. All predicates with this variable as an argument may then still match as if they were true. Thus, the system can continue planning as if the location were known. The only restriction is that no action can be executed that uses an unknown variable as an argument. When such an action is to be executed (e.g., go to *location*1), then the actual instantiation of the variable must be determined before the action is executed (possibly through a perception action). Note that it would be incorrect to continue planning if the truth-values of important predicates depended on the instantiation of the location variable. This tool should only be used when it is appropriate: It is the responsibility of the user not to use the unknown variable if predicates depend on the latter's value.

11.3 Interpreting the input

The sensory system monitoring the execution need not report all predicates that have changed, since many of these may be deduced by SIPE. The system does not plan perception actions to check for additional unexpected predicates, effectively assuming that only the minimal changes consistent with the input and the causal theory have occurred. Alternatively, we could decide on some basis (which would have to be provided as part of the domain-specific description) just how much effort to expend on perception actions to discover other possible unexpected occurrences. For example, if we are told that $(On\ A\ B)$ is not true when we expected it to be, we might want to check to see if B is where we thought it was. As it is, SIPE will simply deduce that B is clear (if no other block is on B) and will not try to execute actions to make further checks with regard to the world. This latter procedure could be very expensive for a mobile robot in the absence of good domain-specific knowledge about what is worth checking.

There is a problem with unexpected effects in deciding how they interact with the effects of the action that was currently being executed (e.g., did they happen before, during, or after the expected effects?). Our solution to this problem is to assume that the action took place as expected and to simply insert a "Mother Nature" action after it that is presumed to bring

about the unexpected effects (including those deduced). The system assumes that any effects of the action being executed that did not actually become true are either provided or can be deduced from the information given. This solution interfaces cleanly and elegantly with the rest of the planner and avoids having to model the way in which the unexpected effects might interact with their expected counterparts.

11.4 The Problem Recognizer

Having just inserted a Mother-Nature node (MN node) in a plan being executed, SIPE must now determine how the effects of this node influence the remainder of the plan. There are two aspects to this: the first involves planning decisions that were based on the effects of this node, and the second involves deductions about the state of the world that were based on those effects.

The second aspect is essentially a truth-maintenance problem. Many effects deduced later in the plan may no longer be true if they depended on predicates that are negated by the MN node. The validity of such deductions must be checked so that the remainder of the plan represents the state of the world accurately. Since it is assumed that processes work as expected whenever their precondition is true and all phantoms that should be protected are true, only deduced effects need to be checked for their dependence on unexpected effects. (The replanner will solve problems having to do with preconditions and phantoms that are not true). Since deduction is not expensive in SIPE (because of the controls described in Chapter 6), the truth maintenance problem is solved simply by redoing the deductions at each node in the plan after an MN node. Even this is avoided in simple cases, because the system carries a list of changed predicates as it goes through the plan; if they all become true later in the plan (without any deduced effects changing in the interim), then the execution monitor need not look at the remainder of the plan (either for redoing deductions or for finding problems).

The problem recognizer finds all problems in the remainder of the plan that might be caused by the effects of the MN node. Since deductions are correctly updated before the problem recognizer is called, it will also find any problems caused by them. The problem recognizer also notices possible serendipitous effects. Because of the rich information content in the plan representation (including the plan rationale), there are only six problems that

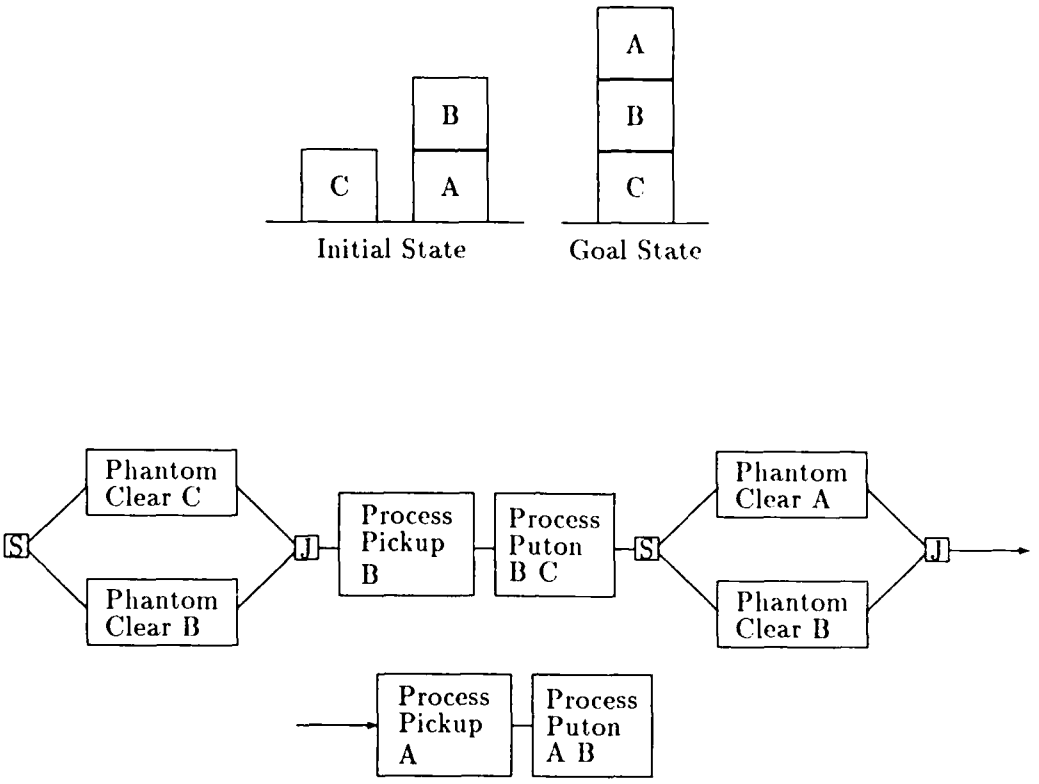


Figure 11.2: Blocks World Problem and Plan

must be checked. All occurrences of the six problems listed below are found by the problem recognizer. These problems constitute the only things that can go wrong with a plan in SIPE after addition of a MN node at the current execution point. The block-world problem in Figure 11-2 will be used to show an example of each type of problem.

• *Purpose not achieved.*

If the MN node negates any of the main effects of the action just executed, there is a problem. The main effects must be reachieved. If, during execution of the first Puton node in the plan in Figure 11-2, either $\neg(On\ B\ C)$ or $(On\ B\ D)$ is given as an unexpected effect, then the MN node inserted after the Puton node will negate the purpose of the Puton node thereby resulting in an instance of this type of problem.

• *Previous phantoms not maintained.*

SIPE keeps a list of phantom nodes that occur before the current execution point (including those on parallel branches), and whose protect-until slot requires their truth to be maintained. If the MN node negates any of these, then there is a problem. The phantoms that are no longer true must be reachieved. Suppose that during execution of the first Pickup node in our example, $\neg(Clear\ C)$ is given as an unexpected effect. This type of problem will then occur, since the phantom node $(Clear\ C)$ has a protect-until slot (not shown in the figure) which effectively points to the first Puton node, but the phantom has been negated by the MN node.

• *Process node using unknown variable as argument.*

If a variable has been declared as unknown, then the first action using it as an argument must be preceded by a perception action for determining the value of the variable. If the B in the example plan were the instantiation of the variable *block*1 (instead of being given as part of the problem), and **unknown block1** were entered during execution of the first Pickup action, then this type of problem would occur with the immediately following Puton action, since it would be applied to an unknown argument.

• *Future phantoms no longer true.*

A phantom node after the current execution point may no longer be true. It must be changed to a goal node so that the planner will try to achieve it. In the sample plan, suppose that $(On\ D\ B)$ were given as an effect during execution of the first Puton node. This type of problem would then occur with the last $(Clear\ B)$ phantom node in the plan, since it would no longer be true when it is expected to be.

• *Future precondition no longer true.*

A precondition node after the current execution point may no longer be true. In this case, we do not want to reachieve it, but rather pop up the hierarchy and perform some alternative action to achieve the goal at that level of the hierarchy. Because the sample plan contains no precondition nodes, we consider an example of this type in the travel-planning domain. Suppose there is an operator for John's taking a taxi to the airport, which has a precondition

that John's car is inoperative. If, during execution of the first part of the plan, SIPE is told that John's car is not broken, this type of problem will occur. In this case the reason for taking a taxi to the airport has been invalidated, and the general replanner will pop up the hierarchy and apply a different operator to get John to the airport (presumably driving his car).

- *Parallel postcondition not true.*

All the parallel postconditions may no longer be true at a join node. (This could be handled by maintaining phantoms, but is more convenient to handle separately.) In this case, we must insert a set of parallel goals after the join node, one for each untrue parallel postcondition. The parallel postconditions of the new join node will be the same as those on the old join node. In the sample plan, the last join node will have both $(On\ A\ B)$ and $(On\ B\ C)$ as parallel postconditions (since they were in parallel originally). Suppose that $(On\ B\ Table)$ were given as an effect during the execution of the last Puton node in the plan. This type of problem would then occur, since the parallel postcondition of $(On\ B\ C)$ would no longer be true.

Because of the way plans are encoded in SIPE, these are the only things that need to be checked when determining whether an MN node affects the remainder of a plan. This illustrates how the rich structure of plans in SIPE helps produce efficient problem detection. However, processes (actions) are assumed to work whenever their precondition is true and when all protected phantoms are true. This should not be a burden on the user, since all such necessary conditions should be encoded as either preconditions or goals, in any case. There is currently no check for loops caused by the same error happening repeatedly, with the same fix being proposed by the general replanner each time. Various simple checks could easily be added if this were a problem.

In addition to the above problems, possible serendipitous effects are also noted and included in the list of problems by the problem recognizer. If the main effect of some action later in the plan is true before the action is executed, then that is noted as a possible place to shorten the plan. (This is discussed in more detail in the next section).

11.5 Replanning Actions

The eight replanning actions implemented in SIPE – Reinstantiate, Insert, Insert-conditional, Retry, Redo, Insert-parallel, Pop-redo, and Pop-remove – provide sufficient power to alter plans in a way that often retains much of the original plan. These are domain-independent actions, and they form the basis of the general replanner and can be referred to by other parts of the system and by domain-specific systems the user might develop for directing error recovery. The first seven actions can all be used to solve problems found by the problem recognizer, while the last is used to take full advantage of serendipitous effects.

Four of the replanning actions change the plan so that it will contain unsolved problems. The intention (see Figure 11-1) is that the plan will then later be given to the normal planning module of SIPE (possibly after a number of these replanning actions have changed the plan). The planner will then attempt to find a solution that solves all the problems that have been corrected in the plan. Any problems in the plan caused by the addition of goals will be dealt with as part of the normal planning process.

- *Insert (node1 node2)*

This action inserts the subplan beginning with node1 (which has been constructed) into the current plan after node2. All links between the new subplan and the old plan are inserted correctly. This is used as a subroutine by many of the actions below.

- *Insert-conditional (variable node context)*

This complements the unknown variable feature — it inserts a conditional around the given node that tests whether the given variable is known. If it is, the given node is executed next; otherwise a failure node is executed.

- *Retry (node)*

The given node is assumed to be a phantom node and it is changed to a goal node so that the planner will perceive it as unsolved.

- *Redo (predicate node context)*

This action creates a goal node whose goal is the given predicate. It then calls Insert to place this new node after the given node in the plan.

• *Insert-parallel (node predicates context)*

This action essentially does a Redo on each predicate in the given list of predicates and puts the resulting goal nodes in parallel, creating new split and join nodes. This subplan is inserted after the given node in the plan. The planner will see these new nodes as unsolved goals. This action is useful for reachieving parallel postconditions.

• *Reinstantiate (predicate node context)*

This action attempts to instantiate a variable differently so as to make the given predicate true in the situation specified by the given node. This appears to be a commonly useful replanning action. For example, it might correspond to using a different resource if something has gone wrong with the one originally employed in the plan, or deciding to return to the hopper for another screw rather than trying to find the one that has just been dropped. It is a complex action that raises many issues that are discussed below.

• *Pop-redo (node predicates context)*

This action and Pop-remove are the most complicated of the replanning actions; it is used to remove a hierarchical wedge from the plan and replace it with a node at the lowest level. Pop-redo is used when a precondition node is no longer true and another action must be applied at a higher level. It could also be used to find higher-level goals from which to replan when there are widespread problems causing the replanning to fail (this is not currently implemented). The removal of a wedge from a plan is discussed in detail in a following section.

• *Pop-remove (node predicates context)*

This action is used to takes advantage of serendipitous effects to shorten a plan. Like Pop-redo it removes a wedge, but this action does not insert a node. However, Pop-remove is more complicated because it is nontrivial to decide which wedge to remove. SIPE's heuristics for this are discussed in the following section on removal of wedges. Briefly, serendipitous effects are exploited only if doing so does not change the rest of the plan

11.5.1 Reinstantiation of Variables

One replanning action reinstantiates variables without changing anything else in the plan. For example, when getting screws from a hopper this procedure may be the correct response when you drop a screw - simply execute the same plan, returning to the hopper to pick up a different screw. However, the general problem is quite complicated. There are any number of constraints and instantiations on the plan variables from different parts of the plan. Reinstantiations involve removing some of these and trying to replace them. However, there are two problems: it is not easy to determine all the consequences that have been propagated from the old instantiation choice (without implementing a truth maintenance system on top of the planner); and in general, you must reinstantiate a whole subset of variables to solve the problem, not just one, and it is difficult to pick the correct subset out of the huge number of possibilities.

Two different solutions to the latter problem have been tried in SIPE. One solution is to choose a set of variables using the following algorithm. Consider the variables in the failed node as possible candidates for reinstantiation. For each one, go up the hierarchy to the point where the variable was first introduced. This determines a wedge that in some sense is either causing or signalling the problem. Consider for reinstantiation only those variables whose instantiations were not forced by choices made inside this wedge. The intuition behind this approach is that because an instantiation was not forced by this wedge, the wedge itself may quite likely work without modification on another instantiation of the same variable. The check for where choices are forced is simple in SIPE, because all constraints (including instantiation constraints) are posted relative to choice points, and it is easy to determine which choice points are in a wedge.

In practice, trying to reinstantiate such a set did not work acceptably, so the system no longer expends effort attempting it. While it may have been a feature of our test problems, there never seemed to be a set whose reinstantiation would create a correct plan. Furthermore, reinstantiating a whole set of variables further exacerbates the first problem of dependencies in the remainder of the plan (as discussed below). The currently implemented algorithm is to look only for reinstantiations of single variables; this is efficient and evidently powerful enough to be useful. The Reinstantiate algorithm loops through the arguments of the predicate given to it. For each argument that is a planning variable (as opposed to an actual ground

instance), Reinstantiate checks to see if there is another instantiation for it that will make the predicate true. This is cheap and efficient in SIPE, since it merely involves removing the instan constraint on the variable from the current context (and also from all variables constrained to be the same as this one), and then calling the truth criterion (which will return possible instantiations) to determine if the predicate is now true. Note that all other constraints that have been accumulated on this variable are left intact, so only instantiations that meet all relevant requirements are found.

Since later parts of the plan may depend on properties of a variable's instantiation, a reinstantiation can potentially introduce a large search space since the plan may turn into a problem to be solved when these dependencies are updated. While such dependencies are minimized by only finding reinstantiations that satisfy all constraints (e.g., all requirements made by later preconditions and phantoms will be satisfied because they are encoded as pred constraints), they still exist, e.g., different deductions may be made, and the global constraint network may not be satisfiable. To prevent the introduction of a search space, Reinstantiate is limited by the requirement that it not introduce new problems. If new instantiations are found, Reinstantiate checks the remainder of the plan to see if any parts of it might be affected by the new instantiation (in part by using the problem recognizer), and accepts only those instantiations that cause no new problems. If all new instantiations are rejected, the old instan constraint is simply replaced.

The implementation described above opts for reinstantiation only when it is likely to be the correct solution. This is consistent with SIPE's running efficiently on the problems it does solve. Alternatively, new instantiations could be accepted even though they caused problems — as long as the problems are less severe than the problems incurred by keeping the old instantiation. Since SIPE has no way of comparing the difficulty of two sets of problems, we do not do this. However, it would not be difficult to change SIPE to explore the search space so introduced if a domain warranted it. There are also ways to partially lift this restriction at the cost of a moderately increased search space (though the tradeoffs involved appear to depend on the domain).

As an example of the use of Reinstantiate, let's consider the above-mentioned problem of dropping a screw. Suppose that *screw*1 is a planning variable, while S1 and S2 are particular screws. The plan being executed could have *screw*1 instantiated to S1, a phantom to be maintained with the goal of (*KnownLoc screw*1), and a process node for moving *screw*1

to achieve (*At screw1 Workbench*). During execution of the latter node, SIPE is told that the finger separation of the arm is zero. From this it could deduce (among other things) ¬(*KnownLoc screw1*) and ¬(*At screw1 Workbench*). The problem of not achieving the purpose of the process node will cause the replanner to insert a goal node in the plan for reachieving (*At screw1 Workbench*). Without Reinstantiate, this would involve finding the location of S1 and moving it to the workbench (since *screw1* is instantiated to S1) — which may be a very hard problem (as anyone who has ever dropped a screw is aware). The problem of not maintaining the phantom node could trigger Reinstantiate on the KnownLoc predicate, which would result in *screw1* being reinstantiated to S2 (whose location is known). This would introduce no new problems, and SIPE could proceed to get a screw at the workbench by getting S2 from the hopper.

11.5.2 Removing Wedges from Plans

When redoing a precondition failure, it is easy to determine the wedge to be removed, since precondition nodes are copied down from one level to another. The top of the wedge to be removed is the node that was expanded to initially place the given precondition node in the plan. However, removing a wedge when attempting to take advantage of a serendipitous effect, as in Pop-remove, is more complicated because it is nontrivial to decide which wedge to remove. We discuss SIPE's algorithm for this choice below.

Removing a wedge, for whatever reason, in practice splices out only the lowest level of the wedge, as planning will continue only from this level. In the Pop-redo case, the subplan that is removed at the lowest level is replaced by a copy of the goal or choiceprocess node that was at the top of the wedge (using the Insert replanning action). This is seen as an unsolved goal by the planner, which automatically checks during further planning whether expansions of this node cause problems later in the plan. There is one potentially serious complication: Various constraints may have been posted on the planning variables because of decisions made in the wedge of the plan that has been [effectively] removed. Fortunately, because of SIPE's use of alternative contexts, it is easily solved. This problem is solved by removing from the current context all the choice points that occurred in the wedge of the plan that was removed. This new context is given as the context argument to future planning actions, and no further action need be taken. This results in ignoring precisely those constraints that

should be ignored.

Let us consider the example mentioned earlier of John planning to take a taxi to the airport when his car is broken. The operator for taking the taxi could have a precondition $\neg(Has\ John\ auto1) \vee (Broken\ auto1)$. (This will match John's not having a car or his car being broken.) This operator is applied to solve the goal node (*At John Airport*) at a high level in the plan, causing a precondition node for the above precondition to be inserted into the plan and copied down to all lower levels of the plan. Suppose that, during execution, $\neg(Broken\ auto1)$ is entered as an unexpected effect during execution of a process before the precondition node. This node is a future precondition which becomes false, and the general replanner will apply Pop-redo to the problem. The wedge that is deleted has the goal node (*At John Airport*) at the top. This may be a very large wedge if its lowest level is as detailed as "find the phone book, look up taxi in the yellow pages, dial a taxi company," etc. At the lowest level, the whole plan of finding a taxi and taking it to the airport is spliced out and replaced by an (*At John Airport*) goal node. When SIPE's planner is later called on this plan, this goal node may be solved by John's driving his car to the airport.

Let us now turn our attention to the problem of choosing a wedge to remove when there is a serendipitous effect. There may be various wedges that are candidates and, as with Reinstantiate, these candidates may cause problems later in the plan if they are removed. Pop-remove currently handles this case in the same way it handles Reinstantiate. Namely, it removes a wedge, checks to see if this causes any problems, and, if there are any, replaces the wedge. Thus, serendipitous effects are exploited only if doing so does not change the rest of the plan. This is a trade-off like the one discussed previously. SIPE again opts for efficiency, but could easily be changed to explore the additional search space of replanning after the removal of wedges.

Pop-remove eliminates any search by generating only one candidate wedge for removal. It gives up taking advantage of the serendipitous effect if this wedge does not work. The candidate wedge is generated by following ancestor links from the node given to Pop-remove (which supposedly has a purpose that has become true serendipitously), as long as some main effect of the candidate node is made true by one of the predicates in the list of given predicates that have unexpectedly become true. The candidate node found in this manner determines the candidate wedge. The wedge is rejected immediately unless all its main effects are true in the given list of predicates.

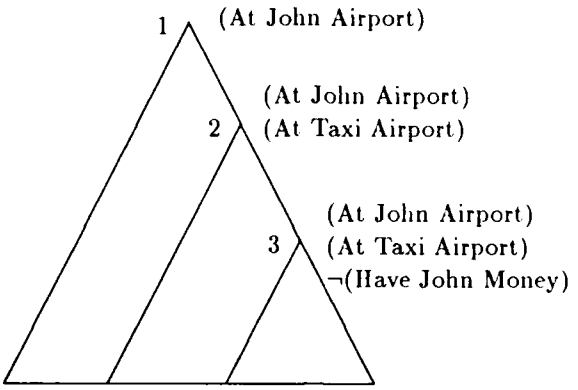


Figure 11.3: Hierarchical Wedges with a Common Last Action

Figure 11-3 uses the example of getting John to the airport to help illustrate this selection process. This example depicts a frequently occurring case in which the last action at one level of a wedge achieves the main effect of every level above that. For example, at Level 1 the goal is only to get John to the airport. At Level 2, after the choice has been made to take the taxi, the last node will achieve getting both John and the taxi to the airport. If Level 3 plans the mechanics of leaving the taxi, the last node there might contain all these higher-level effects as well as the thinner state of John's wallet.

The above selection process requires that all goals generated at a higher level and achieved in the candidate wedge be achieved before the wedge becomes a candidate, while goals generated at a lower level than the top of the candidate wedge need not have been achieved serendipitously. Thus, for Wedge 2 to be selected in Figure 11-3, the serendipitous effects must include (*At John Airport*) from the higher level (as well as (*At Taxi Airport*)) but need say nothing about how much cash John has since that is at a lower level. (It is assumed that, as long as the highest-level goal is achieved, we do not care about the lower-level goals that were necessary to bring this about.) The main effects of higher-level nodes that are achieved within a candidate wedge are easily checked because they are copied down as effects of the node that achieves them. Thus, checking to verify that all main effects of the candidate wedge are true ensures that all important higher-level effects will be true. In the example as shown, Wedge 2 can never be selected by SIPE's algorithm since Wedge 1 will work whenever

Wedge 2 does. However, in another example the effects of Wedge 1 might be achieved at Level 2 before Wedge 2, so that Wedge 2 might then be selected.

11.6 Guiding the Replanning

The replanning actions of the preceding section form the basis for the general replanner, which can be operated interactively as well as automatically. The general replanner takes a list of problems as well as possible serendipitous effects from the problem recognizer, and calls one or more of the replanning actions in an attempt to solve each problem. Before attacking each problem, it first checks that the problem is still a problem, since responses to previous problems may solve many problems at once (e.g., removing a wedge from a plan may remove many problematic nodes).

If the problem is a purpose that is not being achieved, the system tries a Redo, which inserts the unachieved purpose as a goal node after the Mother-Nature node. If the problem is a previous phantom not being maintained, the replanner first tries Reinstantiate and, if that fails, it calls Retry. The idea is that, if there is another object around with all the desired properties, it would be easier to use that object than to reachieve the desired state with the original object. If a process node has an unknown variable as an argument, Insert-conditional is called. If a future phantom is no longer true, Retry is called. As with maintaining phantoms, Reinstantiate may be more appropriate, but, in both cases, this depends entirely on the domain; thus the selection here is arbitrary. For preconditions that are not true, the general replanner first calls Reinstantiate and, if that fails, calls Pop-redo. If parallel postconditions are not true, the general replanner calls Insert-parallel with the appropriate parallel goals.

Once every problem has been addressed according to the above algorithm, the original plan has been modified into a problem that can be given to the standard SIPE planner. There is a whole search space of modified plans that could be given to the planner (e.g., choosing different wedges to remove, etc.). Currently, the modified plan constructed above is the only one in the space of possible modified plans for which the replanner invests effort. If this plan cannot be solved by the planner, the the replanner attempts to solve the original problem in the current situation. The reasons for this are discussed later in this chapter. One major reason is that this search space is combinatoric and it is not even known if the modified plans

are part of a valid solution, while solving the original problem at least will find a solution if one exists.

While a general replanning capability is a significant achievement, one cannot expect very impressive performance from a replanner that does not have domain-specific information for dealing with errors. For example, whether or not Reinstantiate is likely to succeed will be dependent on the domain. The automatic replanner makes reasonable guesses at what might be a good choice in the domains on which SIPE has been tested. Since it merely chooses a replanning action for each type of problem that is found, it is very simple and could easily be rewritten for different domains.

11.7 Examples

This section presents two simple examples of SIPE monitoring the execution of a simple plan, then replanning when things do not go as expected. SIPE has been tested on larger and more complex problems than those presented here. Our examples involve a standard block-world with On and Clear predicates and a Puton operator, as described in Chapter 3. The user inputs only what is explicitly mentioned in boldface below; everything else is generated automatically by the system. The first problem was constructed to show the successful use of the Reinstantiate replanning action, and the second shows how the system inserts a newly created subplan during the replanning process.

Figure 11-4 shows the initial world state and the original problem. The problem is to get A on C in parallel with getting any blue block on any red block. In the initial world B1 and B2 are the only blue blocks (they are both on the table) and R1 and R2 are the only red blocks (R1 is on B1 and R2 is on the table and clear). Since A and C are both clear initially, SIPE finds (in one second) a two-action plan of putting A on C in parallel with putting B2 on R2, as shown in Figure 11-5.

This plan is then given to the execution monitor module of SIPE, which asks if P197 or P168 is to be executed first. The user types **P197** and the system asks for unexpected effects. In this case the user types **(On D R2)** to show one unexpected effect, namely D has suddenly appeared on top of R2. This creates a MN node after P197 which also has the following effects deduced by the system: $\neg(On\ D\ Table) \wedge \neg(Clear\ R2)$. The problem

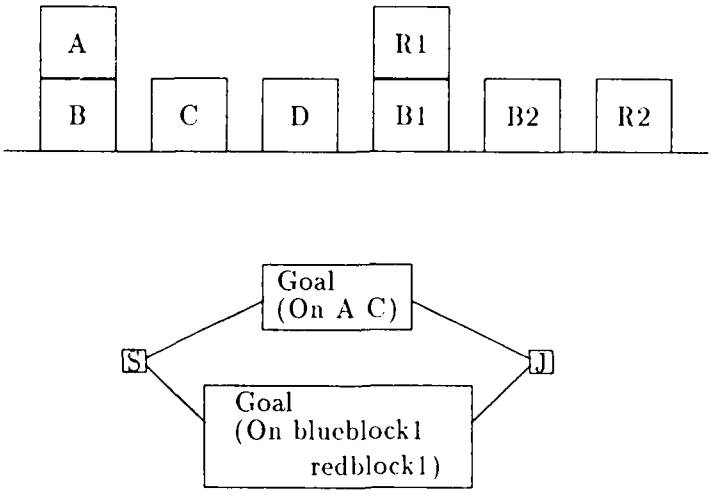


Figure 11.4: Initial Blocks World and Problem to be Solved

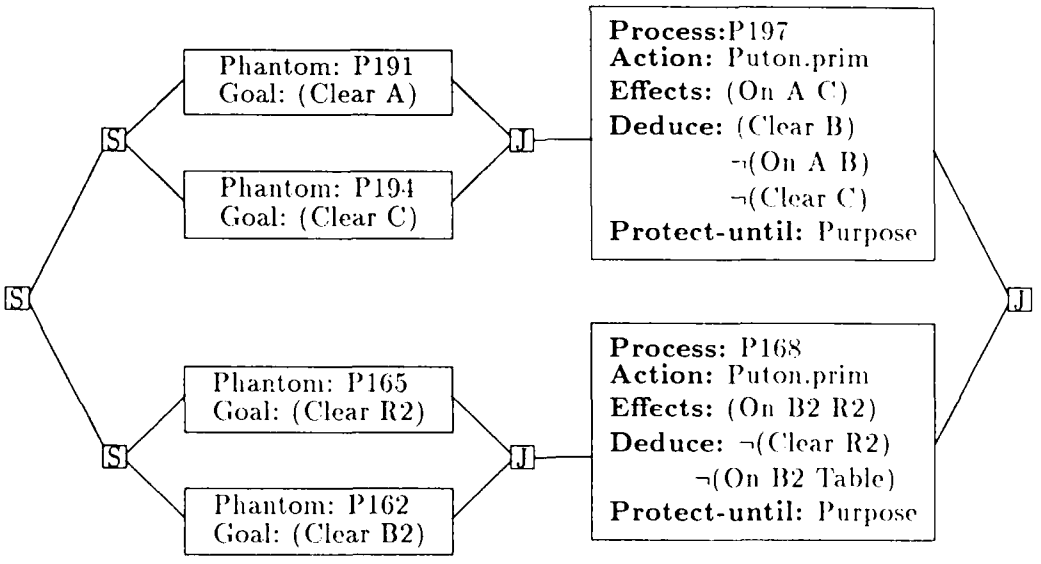


Figure 11.5: Initial Plan Produced by SIPE

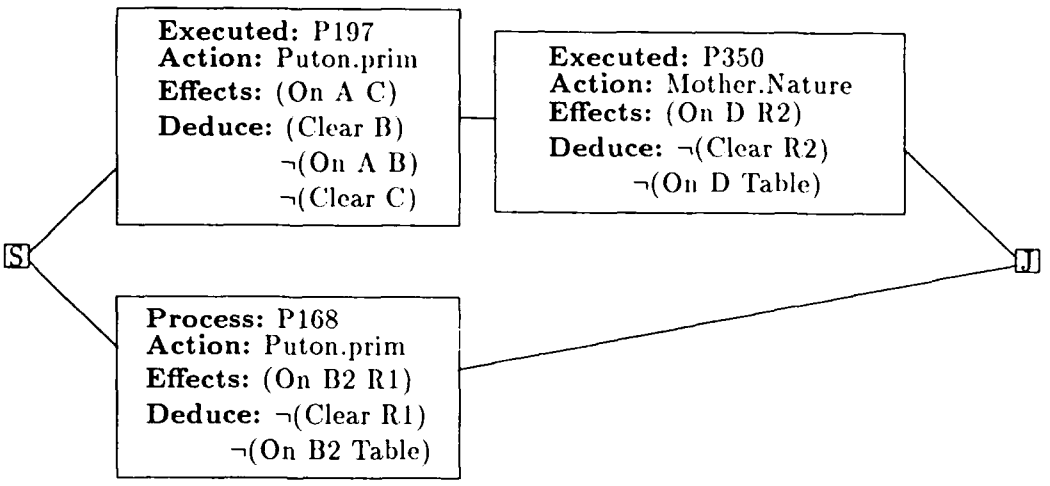


Figure 11.6: New Plan Produced for Continuing Execution

recognizer is called and it finds only one problem, namely the phantom node P165 in a parallel branch was being maintained but is no longer true. This is given to the general replanner which first tries Reinstantiate. This succeeds as the *object*1 variable in the phantom node can be rebound to R1 without causing any new problems in the plan. The plan in Figure 11-6 is passed from the planning module back to the execution monitor (without showing phantom nodes). P168 is then executed without any unexpected effects and the goal is achieved. Note that the original plan was retained in its entirety and that B2 was placed on R1 instead of R2, thus achieving the original goal of getting A on C and any blue block on any red block.

The second problem is the same as the first, except that the variable *redblock*1 is constrained not to be R1 (by specifying `IS NOT R1` in the original problem). The original plan produced by SIPE is the same, as is the unexpected situation input by the user. The problem recognizer again passes the same problem to the general replanner. This time SIPE tries Reinstantiate and fails (since R2 is the only other red block), so it calls Retry, which causes the (*Clear R2*) phantom in Figure 11-5 to be made into a goal. The planner solves this by producing a plan that puts D back on the table before B1 is placed on R2. The subplan shown in Figure 11-7 replaces the reachieved phantom node, on the parallel branch before the *Puton B2 R2* node in Figure 11-5. Without further unexpected events, the plan so constructed then executes correctly to achieve the original goal. Alternatively, more unex-

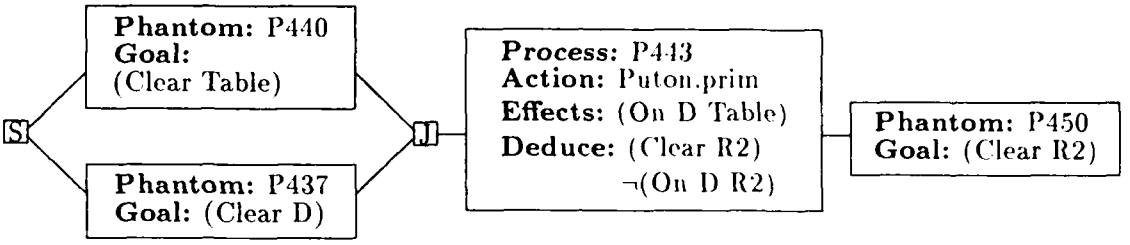


Figure 11.7: Subplan for Replacing Phantom P165

pected occurrences could be given during execution of the newly constructed plan, and SIPE would again go through a similar loop of finding and fixing problems until the original goal is achieved.

11.8 Searching the Space of Modified Plans

SIPE has chosen the course of solving only one modified plan because constantly checking for problems in planning and correcting them is time consuming, the occurrence of problems means less of the old plan is being reused, and there is no guarantee that the plan we are attempting to reuse is part of a valid solution, while starting over guarantees a solution will be found if one exists.

Problems that were encountered in the mobile robot domain motivated the consideration of how replanning might backtrack after the initial attempt at replanning fails. The conclusion we reached in all but the simplest cases is that the process of intelligently deciding where to replan is too expensive to implement. The general replanner could easily be changed if reusing some part of the old plan were a high priority in a particular domain. If the planner fails on one modified plan, the replanner could generate another by applying different replanning actions or removing certain wedges that may be causing problems. In addition it could continue to modify the plan during the planning process. For example, during replanning the planner may fail to find a solution because a future precondition is no longer true. An alternative to quitting after such a failure would be to apply Pop-redo again to this precondition recursively and continue planning.

The following problem indicates some of the difficulties. Suppose the plan calls for the

robot to walk down a hall, go through the first door into a conference room, and pick up the slide projector. Suppose this door is blocked but there is a second door that also leads into the same room. Is it possible to replan to use the second door without redoing the part of the plan that comes after entering the conference room? In general, the problem of navigating to the slide projector will be different since we have a different starting point from within the conference room. The plan of what to do after reaching the slide projector should remain intact, but how can this crucial point in the plan be recognized? In fact, one would like to find the point in the plan where the paths from the two doors to the slide projector joined. Of course, things are not actually this simple — for example, the robot may not be able to reuse any of the original plan if it has used up its batteries going all the way down to the second door.

It is assumed that the planner knows which node at the most primitive level is causing a problem in the plan, called the *failed node*, which in the example might be a node requiring the conference-room door to be unblocked. To replan this failed node, we would like to replace a wedge containing it with the node at the top of the wedge and call the standard planner on the result. If the replanning is to succeed, we must choose an ancestor of the failed node, high enough in the hierarchy so that its wedge will contain everything that must be replanned, thus ensuring that we will produce a correct plan. In our example, we would ideally like to remove a wedge ending at the point where the two paths to the projector joined. Removing a wedge that ended earlier will produce a modified plan that cannot lead to a solution, while removing a wedge ending later will cause the system to replan more than is necessary. As one might suspect, there is no tractable way to guarantee picking the correct wedge. Intuitively, it would seem that determining the correct wedge involves all the same reasoning that solving the original problem entails.

SIPE's solution is to resolve the original problem after attempting replanning on one possible modified plan. Solving the original problem is equivalent to backing up to the largest wedge and replanning it. Searching the space of possible wedges is expensive because each one may be as difficult as the original problem. One of the algorithms we considered for choosing a wedge entailed going back to the first node that had an untried operator, i.e. a node that represents a choice point at which the system has other (as yet untried) choices of which operator to apply. This algorithm does not give the desired results because the replanning often does not need to apply different actions. In the robot domain, it is frequently the case

that the same actions must be done, but in the altered world state used for the replanning, these same actions will result in different instantiations for many variables. The ideal wedge for replanning often requires (in our test domains) applying the same operators that were used in the original plan for a level or two, but then applying different operators at lower levels. Therefore, applying the above-mentioned algorithm often results in selecting a wedge that is not high enough in the hierarchy, which dooms the replanning to failure.

Another attractive algorithm is to return to the node which first introduced the variables that are the arguments of the failed node. In our example, if the failed node involves the conference room door being blocked, then going up the hierarchy to a level at which the door is not mentioned gets us to the goal of getting the robot into the conference room. This is not quite far enough, since the first few actions after entering the conference room need to be replanned. In addition, this algorithm often returns all the way to the top. It may be possible to combine the above two algorithms to obtain a more conservative choice of wedges (in the sense of assuring success in the replanning). Such an algorithm would first go up the hierarchy until all the variables of the failed node disappear, and then continue up until it finds a node with an untried choice.

11.9 Summary

Given correct information about unexpected events, SIPE is able to determine how this affects the plan being executed. In many cases, it is able to retain most of the original plan by making changes in it to avoid problems caused by these unexpected events. It is also capable of shortening the original plan when serendipitous events occur. It cannot solve difficult problems involving drastic changes to the expected state of the world, but it does handle many types of small errors that may crop up frequently in a mobile robot domain. The execution-monitoring package does this without the necessity of planning in advance to check for such errors.

Very few classical planners address the execution monitoring and replanning problem. Those that do have replanners that are considerably simpler and less powerful than that of SIPE [37]. They also do not allow the input of arbitrary predicates as SIPE does, so the general replanning problem never arises. One major contribution of this work is the development of a general set of replanning actions. In addition to their centrality in the replanner, they

can be used elsewhere in the system. The plan critics use them to modify plans, producing more optimal solutions to problems in the process. One can envision encoding error recovery knowledge by using these replanning actions.

These actions provide sufficient power to alter plans in a way that often retains much of the original plan. The success of these mechanisms can largely be attributed to taking advantage of the rich structure of SIPE's planner and its plans. The replanner takes advantage of the efficient truth criterion to discover problems and potential fixes quickly, and applies the deductive capabilities to provide a reasonable solution to the truth maintenance problem. The fixes suggested by the replanner need involve only the insertion of new goals into the plan, since calling the planner as a subroutine will solve these goals in a manner that assures there will be no conflicts with the rest of the plan. SIPE's execution-monitoring capabilities make extensive use of the explicit representation of plan rationale. The problem detector makes uses of the information encoded in protect-until slots, phantoms, and preconditions to quickly find all the problems with a plan. Furthermore, it does not remove parts of the original plan unless the parts are actually problematical. The replanning actions make use of constraints and contexts whenever they consider removing part of the plan.

From the beginning, the rationale behind SIPE has been to place enough limitations on the representation so that planning can be done efficiently, while retaining enough power to still be useful. This motivation underlies most of the design decisions that have been made in implementing the replanning module. For example, Reinstantiate and Pop-remove are limited to prevent the exploration of large search spaces. The major limitations of this research stem from the assumption of correct information about unexpected events. This avoids many difficult problems, the most important of which is generating the high-level predicates used by SIPE from information provided by the sensors. This appears to be the most critical issue in getting a high-level planner such as SIPE to control a mobile robot. Part of the problem is heuristic adequacy — the robot cannot wait ten minutes for a vision module to turn pixels into predicates while the world is changing. Rosenschein's situated automata approach holds promise for being able to solve the pixels-to-predicates problem in a heuristically adequate manner [25].

Chapter 12

Summary

SIPE has extended the classical AI planning paradigm farther than any other system, addressing the balance between epistemological and heuristic adequacy over several years of development. It extends the classical approach in several novel ways, the most important being reasoning about resources, posting constraints, providing a general replanning capability, and using a deductive causal theory to deduce context-dependent effects. While SIPE builds upon classical AI planning work, its representations and algorithms have almost nothing to do with earlier systems because of the complications introduced by the above features.

As we have seen, the problem addressed by extended classical AI planners involves several combinatorial problems. These include the truth criterion, the unification problem (with reasonably powerful constraints on variables), the problem of parallel interactions, the resource allocation problem, the search through the space of possible plans, and the search through modified plans during replanning. Throughout this report we have stressed the heuristics, algorithms, and limiting assumptions applied by SIPE to avoid these combinatorial explosions. This has resulted in an efficient system that provides a useful planning capability, as evidenced by the examples and runtimes given in this report.

These heuristics and algorithms address problems that must be addressed by all planning systems. We briefly summarize these techniques here to provide a more unified view of how heuristic adequacy is accomplished. The heart of the system, the unification algorithm and truth criterion, is based on heuristics. The unification algorithm avoids the combinatorial problems introduced by the use of constraints by doing a complete check of constraints only at

the top level of recursive calls. The truth criterion includes several heuristics and algorithms for avoiding combinatorial problems and improving efficiency. These include the posting of pred and not-pred constraints, the implementation of quantifiers, and methods for handling nonlinearity. The latter are some of the most important heuristics from the standpoint of efficiency, but, unfortunately, also some of the most limiting.

Deductive causal theories allow deduction of context-dependent effects in an efficient manner. SIPE's algorithm controls the triggering of domain rules and efficiently computes the deductive closure of these rules. One heuristic is used to avoid a combinatorial search space: the system will constrain variables in an attempt to match a domain rule, but only when the two variables are already constrained to be of the same class.

Like other classical planners, SIPE exhibits hierarchical promiscuity by mixing abstraction levels. Unlike previous planners, SIPE recognizes problems caused by this and provides various solutions. The system provides the option of the ABSTRIPS solution which to impose a depth-first, left-to-right planning order that is sensitive to change in abstraction level. In addition, it provides for delayed application of operators when necessary, but permits expediential planning in other cases.

No previous classical planner has incorporated a general replanner. Since optimal recovery from an arbitrary error poses an intractable problem, SIPE again relies on heuristics during replanning. There is a large space of different ways to modify a plan. SIPE has chosen the course of trying only one modified plan because (1) constantly checking for problems in planning and correcting them is time consuming, (2) the occurrence of problems means less of the old plan is being reused, and (3) there is no guarantee that the plan we are attempting to reuse is part of a valid solution, while starting over guarantees a solution will be found if one exists. After trying several algorithms for exploring the space of modified plan, we concluded that, in all but the simplest cases, intelligently deciding where to replan is too expensive to implement.

The plan critics use heuristics that are too numerous to mention in detail. SIPE provides several heuristics for proposing linearizations and phantomizations (e.g., when there is only one way to phantomize something then the phantomizations is proposed), as well as heuristics for ordering a set of proposed linearizations. One mechanism the system uses effectively is its analysis of a proposed linearization by the replanning actions. Using the full power of the

problem recognizer is a powerful way to determine problems caused by linearizations. Furthermore, replanning actions are used to solve some of the problems so detected which both corrects invalid plans and makes suboptimal plans more efficient. The power to modify plans in this way makes the system considerably more powerful than previous classical planners.

Chapter 13

Publications

Several articles describing our research have appeared in journals and conference proceedings over the last several years. This report gathers all of this information in one place, but synthesizes it in a novel way, describes new capabilities that have not yet been described in the literature, and updates many outdated descriptions in earlier papers. The publications listed below, as well as many talks given over the years by Dr. David E. Wilkins at various workshops and conferences, have assured a high degree of visibility for the research described in this report.

Dr. Wilkins also participated in other activities at SRI expense that helped increase awareness of the research described here. During the past year, he attended the DARPA planning workshop in Washington, D.C., and chaired the panel on strategic planning, making a presentation on the current state of planning research and directions for future research. He was also a member of the program committee for the 1987 AAAI conference.

- Journal articles describing work on this project (all by Dr. David E. Wilkins):

  "Causal Reasoning in Planning", *Computational Intelligence 4*, 1988, forthcoming.

  "Recovering from Execution Errors in SIPE", *Computational Intelligence 1*, 1985, pp. 33-45.

  "Domain-independent Planning: Representation and Plan Generation", *Artificial Intelligence 22*, April 1984, pp. 269-301.

- Papers in conference proceedings (all by Dr. David E. Wilkins except as noted):

  "Hierarchical Planning: Definition and Implementation", *Proceedings of the Seventh ECAI*, Brighton, England, 1986, pp. 466-478.

  "Representation in a Domain-Independent Planner", *Proceedings of IJCAI 83*, Karlsruhe, Germany, 1983, pp. 733-740.

  "Parallelism in Planning and Problem Solving: Reasoning About Resources", *CSCSI Conference Proceedings*, Saskatoon, Saskatchewan, 1982, pp. 1-7.

  "Representing Knowledge in an Interactive Planner", by D.E. Wilkins and Ann Robinson, *Proceedings of the AAAI 80*, Stanford, California, 1980, pp. 148-150.

Bibliography

[1] Agre, P., and Chapman, D., "Pengi: An Implementation of a Theory of Activity", *Proceedings AAAI-87*, Seattle, Washingtion, 1987, pp. 268-272.

[2] Allen, J., "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, November 1983, v. 26 n. 11, pp. 832-843.

[3] Bresina, J., "An Interactive Planner that Creates a Structured, Annotated Trace of its Operation", Tech. Report CBM-TR-123, Department of Computer Science, Rutgers University, December 1981.

[4] Chapman, D., "Planning for Conjunctive Goals", Technical Report MIT-AI-TR-802, MIT Laboratory for Artificial Intelligence, Cambridge, Massachusetts, 1985.

[5] Dean, T., "Temporal Imagery: An Approach to Reasoning about Time for Planning and Problem Solving", Technical Report 433, Yale University Computer Science Department, 1985.

[6] Fikes, R., Hart, P., and Nilsson, N., "Learning and Executing Generalized Robot Plans", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 231 249.

[7] Georgeff, M. P., "Actions, Processes, and Causality", *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Timberline Lodge, Timberline, Oregon, 1987, pp. 99-122.

[8] Georgeff M., Lansky A., and Schoppers M., "Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot", Technical Note 380, SRI International Artificial Intelligence Center, Menlo Park, California, 1986.

[9] Goad, C., "Fast 3D Model-Based Vision", in *From Pixels To Predicates: Recent Advances In Computational And Robotic Vision*, Editor, A. Pentland, Ablex.

[10] Halpern, J. Y. and Moses, Y. O., "A Guide to the Modal Logics of Knowledge and Belief", *Proceedings IJCAI 85*, Los Angeles, California, 1985, pp. 50-61.

[11] Hanks, S., and McDermott, D., "Default Reasoning, Nonmonotonic Logics, and the Frame Problem", *Proceedings AAAI 86*, Philadelphia, Pennsylvania, 1986, pp. 328-333.

[12] Hayes, Philip J., "A Representation for Robot Plans", *Proceedings IJCAI-75*, Tbilisi, USSR, 1975, pp. 181-188.

[13] Hayes-Roth, B. and Hayes-Roth, F., "A Cognitive Model of Planning", *Cognitive Science 3*, 1979, pp. 275–310.

[14] Hobbs, J., "Granularity", *Proceedings IJCAI-85*, Los Angeles, California, 1985, pp. 432-435.

[15] Kaelbling, L., "REX Programmer's Manual" Technical Note 381, SRI International Artificial Intelligence Center, Menlo Park, California, 1986.

[16] Ladkin, P., "Primitives and Units for Time Specification", *Proceedings AAAI-86*, Philadelphia, Pennsylvania, 1986, pp. 354-359.

[17] Lifschitz, V. 1987 "On the Semantics of STRIPS", *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Timberline Lodge, Timberline, Oregon, pp. 1-9.

[18] McCalla, G., and Schneider, P., "The Execution of Plans in an Independent Dynamic Microworld", *Proceedings IJCAI-79*, Tokyo, Japan, 1979, pp. 553-555.

[19] McCalla, G., and Schneider, P., "Planning in a Dynamic Microworld", *Proceedings CSCSI Conference*, Saskatoon, Saskatchewan, 1982, pp. 248-255.

[20] McCarthy, J., "Circumscription: A Nonmonotonic Inference Rule", *Artificial Intelligence 13*, 1980, pp. 27-40.

[21] McDermott, D., "A Temporal Logic for Reasoning about Processes and Plans", *Cognitive Science 6*, pp. 101-155.

[22] Pednault, E. P. D., "Synthesizing Plans that Contain Actions with Context-Dependent Effects", *Computational Intelligence 4*, 1988, forthcoming.

[23] Pentland, A.P., and Fischler, M.A., "A More Rational View of Logic or, Up Against the Wall, Logic Imperialists!", *AI Magazine*, Vol. 4, No. 4, pp. 15-18 (1983).

[24] Rosenschein, S., "Plan Synthesis: A Logical Perspective", *Proceedings IJCAI-81*, Vancouver, British Columbia, 1981, pp. 331-337.

[25] Rosenschein, S., "Formal Theories of Knowledge in AI and Robotics", Technical Note 362, SRI International Artificial Intelligence Center, Menlo Park, California, 1985.

[26] Sacerdoti, E., "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence 5 (2)*, 1974, pp. 115-135.

[27] Sacerdoti, E., *A Structure for Plans and Behavior*, Elsevier, North-Holland, New York, 1977.

[28] Shoham, Y., "What is the Frame Problem?", *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Timberline Lodge, Timberline, Oregon, 1987, pp. 83-98.

[29] Stefik, M., "Planning and Metaplanning", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 272-286.

[30] Sussman, G.J., *A Computer Model of Skill Acquisition*, Elsevier, North-Holland, New York, 1975.

[31] Tate, A., "Generating Project Networks", *Proceedings IJCAI-77*, Cambridge, Massachusetts, 1977, pp. 888-893.

[32] Tate, A., "Goal Structure, Holding Periods and Clouds" *Proceedings of the 1986 Workshop on Reasoning about Actions and Plans*, Timberline Lodge, Timberline, Oregon,1987, pp. 267-277.

[33] Vere, S., "Planning in Time: Windows and Durations for Activities and Goals", Jet Propulsion Lab, Pasadena, California, November 1981.

[34] Waldinger, R., "Achieving Several Goals Simultaneously", in *Readings in Artificial Intelligence*, Nilsson and Webber, eds., Tioga Publishing, Palo Alto, California, 1981, pp. 250–271.

[35] Wilensky, R., "Meta-planning", *Proceedings AAAI-80*, Stanford, California, 1980, pp. 334-336.

[36] Wilkins, D. E., "Domain-independent Planning: Representation and Plan Generation", *Artificial Intelligence 22*, April 1984, pp. 269-301.

[37] Wilkins, D. E., "Recovering from Execution Errors in SIPE", *Computational Intelligence 1*, 1985, pp. 33-45.

[38] Wilkins, D. E., "Using Patterns and Plans in Chess" *Artificial Intelligence 14*, 1980, pp. 165-203.

END

DATE

FILMED

8-88

DTIC